2006

Model-based risk assessment

Walid M. Abdelmoez
West Virginia University

Model-based Risk Assessment

Walid M. Abdelmoez

Dissertation submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Engineering

Hany H. Ammar, Ph.D., Chair
Katerina Goseva-Popstojanova, Ph.D., Co-Chair
Gerald Hobbs, Ph.D.
Ali Mili, Ph.D.
Tim Minzies, Ph.D.

Lane Department for Computer Science and Electrical Engineering

Morgantown, West Virginia
2006

Keywords: reliability-based risk, maintainability-based risk, software architecture

ABSTRACT

Model-based Risk Assessment

Walid M. Abdelmoez

In this research effort, we focus on model-based risk assessment. Risk assessment is essential in any plan intended to manage software development or maintenance process. Subjective techniques are human intensive and error-prone. Risk assessment should be based on architectural attributes that we can quantitatively measure using architectural level metrics. Software architectures are emerging as an important concept in the study and practice of software engineering nowadays, due to their emphasis on large-scale composition of software product, and to their support for emerging software engineering paradigms, such as product line engineering, component based software engineering, and software evolution.

In this dissertation, we generalize our earlier work on reliability-based risk assessment. We introduce error propagation probability in the assessment methodology to account for the dependency among the system components. Also, we generalize the reliability-based risk assessment to account for inherent functional dependencies.

Furthermore, we develop a generic framework for maintainability- based risk assessment which can accommodate different types of software maintenance. First, we introduce and define maintainability-based risk assessment for software architecture. Within our assessment framework, we investigate the maintainability-based risk for the components of the system, and the effect of performing the maintenance tasks on these components. We propose a methodology for estimating the maintainability-based risk when considering different types of maintenance. As a proof of concept, we apply the proposed methodology on several case studies. Moreover, we automate the estimation of the maintainability-based risk assessment methodology.

Dedication

To my family with love,

To my mentors with gratitude

Acknowledgements

Thanks to Allah, Most Gracious, and Most Merciful. I would like to express my appreciation and gratitude to my advisor Professor Dr. Hany Ammar. I am thankful for his guidance, support and patience. I was honored to have Dr. Katerina Goseva-Popstojanova co-advise my Ph.D. research, it has been a great experience working with her. I appreciate her patience, humbleness and friendliness. I would like to express my thanks and gratitude to Dr. Ali Mili for his continuous support and encouragement. It has been a pleasure and a privilege to work with him. I am also very grateful to Dr. Gerald Hobbs and Dr. Tim Minzies for being valuable members of my examination committee.

I thank all my West Virginia University research colleagues: Dr. Diaa Eldin Nassar, Dr. Ahmed Hassan, Dr. Vittorio Cortellessa, Dr. Mark Shereshevsky, Rabieh Elkradely, Rajesh Gunnalan, Kalaivani Appukkutty, Tianjian Wang, Ajith Guedem, Venu Dalta, Israr Shaik, Khader Sheik and Sweatha Reddy

I am indebted to my great friends: Mohamed Hussein, Walid Moustafa, Hossam AdelBari, Ahmed Elsherif, Walid AbdelHaleem, Shady Koriatam, Tamer Saber, Magdy El-Batouty, Ayman Abazza, Dr. Mohamed Salem, Dr. Khalid Elmorsy, Hatem Elkanifati and to my friends and colleagues in the Arab Academy for Science and Technology for their great support and valuable advice.

My family has always been a great source of support and love; I really cannot thank each of my family members enough. I am very thankful to my Father Dr. Mohammad Rabie Abdelmoez, my Mother Dr. Samia Farag, and my brother Tarek. I am also very grateful to my Father-in-law Mr. Saber Moustafa , my late Mother-in-law Mrs. Fatheya Shahein and my brother-in-law Emad Moustafa.. And last but by no means least; I am truly grateful for my great wife Rania for being caring, supportive, and considerate ; and my lovely son Youssof and daughter Yosr for being a source of delight, pleasure and hope.

This work is supported by the National Science Foundation through ITR program and by NASA through a grant from the NASA Office of Safety and Mission Assurance (OSMA) Software Assurance Research Program (SARP) managed through the NASA Independent Verification and Validation (IV&V) Facility, Fairmont, West Virginia.

Table of Contents

List of Figures

List of Tables

1 Introduction

To plan the development for a software system, the project manager should assess the risks facing the development effort. Domain experts' subjective decision is inherent to several risk assessment techniques. Thus, they require intensive human involvements and are error-prone. Therefore, risk assessment is supposed to rely on system attributes that we can quantitatively measure from its models. Model-based assessment is gaining importance because it enables us to quantitatively evaluate the attributes of a system. In this dissertation, we focus on model-based risk assessment derived from the system architecture.

1.1 Overview

A sound architecture is the key to build a software system with high quality attributes. Software architecture explicates the structure of the system in terms of components and interactions among them to accomplish the desired requirements. Furthermore, it supports many software development paradigms such as COTS-based software development, product line engineering and component based software engineering. In [Shaw 1995], Shaw was the first to advocate the shift from functional view of software development to architectural view which has been widely embraced since. As architecture became a more significant artifact in developing software systems, the need to quantitatively analyze the architecture has become eminent. The architecture quantitative analysis should reflect its pertinent quality attributes and help us to predict the quality of the software products instantiated from it.

According to NASA-STD-8719.13A standard [NASA 1997] risk is a function of the anticipated frequency of occurrence of an undesired event and the potential severity of resulting consequences. This standard defines several types of risk, such as for example availability risks, acceptance risk, performance risk, cost risk, schedule risk, etc. Software risk management concentrates on developing a product with better quality attributes such as reliability, performance, maintainability, and the uncertainty associated with the product development. It helps project managers in avoiding unpredicted catastrophic problems. Also, it prevents wrong allocation of resources and taking decisions without proper knowledge or adequate information on anticipated future consequences [SEI 2005]. To manage software development projects, managers and developers should rely on processes, methods and tools to facilitate assessment, prioritization and mitigation of various risk aspects. Therefore, risk assessment is an essential part in the management of software development.

In this research effort, we are concerned with model-based risk assessment, which includes reliability-based risk and maintainability-based risk. Reliability-based risk takes into account the probability that the

software product will fail in the operational environment and the consequences of that failure. While, maintainability-based risk takes into account the probability that the software product will need to endure a certain type of maintenance and the consequences of performing this maintenance on the system.

1.2 Background

This Dissertation is related to several areas in the field of software engineering. The main theme works around model-based risk assessment for software architecture modeled using Unified Modeling Language (UML). The major contribution is in the field of reliability-based and maintainability-based risk assessment methodologies. The following sections give a basic background on the recent work in these fields.

1.2.1 Software Architecture

Abstracting the software system to highest level obtain us its architecture. As the size and the complexity of the software systems increase, the need for structuring and organizing it into components increases. As a result, the discourse of software system's architecture becomes essential [SEI 2005]. Software architecture is an important asset because of the following:

- The architecture can be used for communication purposes, as it provides an understandable abstraction by stakeholders, not only software developers but also users and managers.
- Early in the development process of new software, architecture can be available for early analysis of the system's properties.
- Existing systems that evolve can be analyzed at the architectural level to provide a foundation for further development.

1.2.2 The Unified Modeling Language

As software systems become more complex, modeling them to guide development or to help maintenance becomes essential. System models are used to document the analysis and design and to communicate the system artifacts among development and maintenance teams. Therefore, to have a modeling language standard is an important factor for the success of an application development and maintenance. The Unified Modeling Language UML has become the de-facto standard for building Object-Oriented software. UML unified the efforts of Booch [Booch+ 1999], Rumbaugh [Rumbaugh+ 1997], and Jacobson [Jacobson+ 1992]. That effort has matured into UML becoming an OMG (Object Management Group) standard [OMG 2005]. Adopting UML as a standard is motivated by:

* It is programming languages independent.
* It provides a rich language for visual modeling to develop and communicate meaningful models.
* It integrates lots of efforts over the years and blends many models developed.
* It provides the means to extend and to specialize the core concepts.

1.2.2.1 UML Definition:

According to OMG specification: [OMG 2005]

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

It is important to note that UML is a 'language' for specifying and not a procedure or method. The UML is used to define a software system; to detail the system artifacts, to document and construct. It is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology but in itself it does not specify that methodology or process.

1.2.2.2 UML Models

A single model cannot capture static and dynamic system properties. The used models influence how to tackle the problem and how to come up with an appropriate solution. Therefore, complex systems should be analyzed by examining independent views. Static models define the static architecture of the system. They are used to model the elements that make up a system - the classes, objects, interfaces and physical components. Furthermore, they are used to model the relationships and dependencies among the elements of the system. Class diagrams, Package diagrams, Component diagrams, and Deployment diagrams are some of the static views of the system. Dynamic models define the interaction among the system elements to accomplish a system behavior. They contain events, responses, messages, and invocations. Use-Cases, Sequence and Collaboration diagrams, and State-charts diagrams are some of the dynamic views of the system. The following summarizes the modeling diagrams supported by UML [UML 2005]:

- *Class diagrams*: A class diagram defines the basic building blocks of a model: the types, classes and general materials that are used to construct a full model. They depict possible classes and their

relationships. Details of the design are communicated through detailed class diagrams, which include the attributes and the methods of the classes.

- *Package diagrams:* are used to divide the model into logical containers or 'packages' and describe the interactions between them at a high level.
- *Implementation diagrams*:
  - *Component diagram:* are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface.
  - *Deployment diagram:* show the physical disposition of significant artifacts within a real-world setting. Deployment diagrams are related to component diagrams in that a node typically encompasses one or more components.
- *Use Case diagrams*: describe the boundary and interaction between the system and users. They conform in some regards to a requirements model. A use case designates a situation in which the system is used. It defines the system inputs, actions and possible outputs. Use cases are analyzed to construct possible scenarios.
- *Behavior diagrams*: capture the varieties of interaction and instantaneous state within a model as it 'executes' over time.
  - *State-chart diagram*: State-charts are used to model the behavior of complex systems [Harel88]. State charts describe the states or conditions that classes assume over time
  - *Interaction diagrams*: They include sequence diagrams and collaboration diagrams
    - *Sequence diagrams*: show the sequence of messages passed among objects using a vertical timeline. A sequence diagram reflects a scenario of interactions in the system to manifest a use case of the system. Normally, there are one or more scenarios for each use-case.
    - *Collaboration diagrams*: are another view of scenarios. They show the network and sequence of messages between objects at run-time during a collaboration instance.

1.2.3 Reliability-Based Risk Assessment

Risk assessment is an essential part in the management of software development. Performing it in the early phases of software development can enhance allocation of resources within the software lifecycle. Also, it provides useful means for identifying potentially troublesome software components that require careful development and allocation of more testing effort. We are concerned with reliability-based risk, which takes into account the probability that the software product will fail in the operational environment and the consequences of that failure.

In [Yacoub+ 1999], Yacoub et. al. defined dynamic metrics that include dynamic complexity and dynamic coupling to measure the quality of software architectures. Their approach was based on dynamic execution of UML state chart specification of a component and the proposed metrics were based on simulation reports. In [Yacoub+ 2002], Yacoub et. al. combined severity and complexity factors to develop heuristic risk factors for the components and connectors. Based on scenarios, they developed component dependency graph that represents components, connectors, and probabilities of component interactions. The overall system risk factor as a function of the risk factors of its constituting components and connectors was obtained using the aggregation algorithm.

In [Goseva-Popstojanova+2003], Goseva-Popstojanova et. al. proposed a methodology for risk assessment based on the UML specifications such as use cases and sequence diagrams that can be used in the early phases the software life cycle. This risk assessment methodology was entirely based on the analytical methods. First, components and connectors dynamic risk factors were estimated analytically based on the information from UML sequence diagrams. Then, a Markov model wass constructed for estimation of each scenario risk factor and closed form exact solutions are derived for the scenarios, use cases, and overall system risk factors.

1.2.4 Software Maintenance

According to IEEE Standard for Software Maintenance [IEEE 1998], software maintenance is defined as follows:

> *"Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment".*

Other definitions for software maintenance were listed in [Pigoski 1996]. Pigoski concluded that any change activity to the software product after being accepted by the client is maintenance.

Thus, software maintenance is concerned with error corrections and system changes as requirements and environment change. Software maintainers usually are not involved in the original software development cycle generally. They must learn how a program functions before they can change it. The status of system documentation, programmer skill and experience and the attributes of the system itself are some of the variables that affect the maintenance process.

1.2.4.1 Types of Software Maintenance

In "That Maintenance Iceberg" [Canning 1972], Canning summarized how maintenance is categorized. According to Canning, maintenance can be considered in narrow sense as to correct errors

and in wide sense as to extend the functionality of the software and to accommodate changes in the underlying system software or hardware. In [Swanson 1976], E. Burton Swanson offered a typology that account for what was seen as the cause or purpose of the maintenance, or why was the maintenance to be done. In [Lientz+ 1980], authors categorized three types of software maintenance:

- **Perfective maintenance:** to perfect the system in terms of its performance, processing efficiency, or maintainability,
- **Adaptive maintenance:** to adapt the system to changes in its data environment or processing environment, and
- **Corrective maintenance:** to correct processing, performance, or implementation failures of the system.

This categorization was adopted by the IEEE Standard for Software Maintenance [IEEE 1998]. There is a fourth type of software maintenance mentioned in the IEEE Standard for Software Maintenance's Annex A.

- **Preventive maintenance:** to prevent system problems before they occur

Close examination of preventive maintenance reveals that it is not a well established and understood discipline. There is a lot of confusion regarding its definition, scope and meaning [Chapin 2000].

1.2.4.2 Software Maintenance Risks

Many types of risk are ushered when software systems undergo maintenance. They are akin to those we face when developing new software systems, but with different level of risk. Also, maintainers often interact with complex and difficult to comprehend systems, which introduce other types of risk that distinguish the software maintenance process. These types of risk are [Sherer 1997]:

- *Project risk*— Maintenance project cannot be carried out within budget or on time, no effective maintenance process and lack of personnel and maintenance capabilities.
- *Usability risk*— Systems will cause problems and failures after the maintenance is conducted. Usability risk includes functionality, performance, financial and software failure risk.
- *Maintainability risk*— It will be difficult to maintain the system in the future because of the way we conducted this maintenance.

1.2.4.3 Maintainability

According to [Pigoski 1996], the cost of software maintenance averages from 60% to 80% of the overall software system cost. As a result, maintainability is an important software quality factor. The effect of good maintainability of a system is realized in the maintenance stage. It is justifiable to make an investment in software maintainability if there will be a reward in terms of productivity in the maintenance stage.

Software maintainability is an attribute that reflects the ease of performing maintenance to the software product. IEEE definition of software maintainability is as follows [IEEE 1990]:

> *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment".*

Software maintenance cost can be significantly reduced if maintainability is integrated early on in the software development lifecycle. To delay considering maintainability as a goal of the development effort, till reaching the operation phase of the software product, affects its quality and causes the maintenance cost to increase. Software development process, documentation and program comprehension considerably affect maintainability [Pigoski 1996]. It would be useful to quantify the relative magnitude of the risk associated with a maintenance task through an analysis of the system artifacts and the maintenance tasks required.

Furthermore, in recent years the use of commercial software components has increased to cut the cost of software development. Without internal modification, the developer integrates the COTS software into the system that will affect the overall maintainability. This is because the evolution and maintenance of the software cannot be fully controlled by the users. Moreover, the quality of the COTS software documentation and vendor support is immature [Päivi+ 2002]. To achieve a faster and more efficient software development process [William+ 2002], it is highly recommended to use standards with proven techniques and notations such as Unified Modeling Language UML. Several problems result from the lack of communication and integration. One of these problems is implicit inconsistency caused by making changes to the UML model or the system design. This will significantly lead to an increase of the cost and complicate the maintenance process of such systems.

1.2.5 Software Design Patterns

In Merriam-Webster dictionary, some of the definitions of the word *pattern* are *"a form or model proposed for imitation"* or *"something designed or used as a model for making things"*. Thus, when we use the word pattern we are reflecting on how we do something or how we pursue our intent. In a mature

craft, it is common to find an archive of best practices that describes effective methods for achieving aims or solving problems in different context. Also in any community of practitioners, you find that they invent jargons to help them explain their intent or how to accomplish that intent. These jargons usually refer to patterns, or standardized ways to do things. As the community becomes more mature, they try to document these patterns in order to standardize the way things are done. This documentation tries to convey the accumulated wisdom to next generations.

The architect Christopher Alexander was the first to document a craft's best practices. Though, his patterns relates to architecture of building, not software. He described patterns for successful towns, buildings, and room designs. He developed a prototype of pattern books from his work in pattern cataloging for architecture [Alexander+ 1977, Alexander 1979]. He developed the fundamental template of a pattern, as a Context-Problem-Solution.

The software community adopted Alexander ideas; Kent Beck and Ward Cunningham introduced the idea of patterns to software development. They wrote the first set of patterns that were addressing user interfaces [Beck+ 1994] [Cunningham 1994] [Beck+ 1996]. Erich Gamma's 1991 doctoral thesis [Gamma 1991] was the first published work about patterns in software development. The *Hillside Group* was formed to explore patterns further and promote their use in software development. In 1994, they founded the first PLoP (*Pattern Languages of Programs*) conference.

The 'Gang of Four' book [Gamma+ 1995] presented the first well-described and documented catalog of design patterns for object-oriented designs. It presented well-used and known design solutions for object-oriented development. They documented a set of twenty-three patterns. They classified them under *Behavioral*, *Structural*, and *Creational*. The use of patterns at the architectural level of software development is introduced in The "Gang of Five" book [Buschmann+ 1996]; they classified the software patterns as architectural patterns, design patterns and idioms. Nowadays, the main sources of pattern evaluation and cataloging are the *Pattern Language of Programs* conferences.

Design patterns can help improve the software development because they:

- Provide a common vocabulary for the designs, which helps in understanding it
- Provide abstractions for the system, which reduces its complexity.
- Are proven building blocks from that help in building more complex applications.
- Help producing new designs as they are used as guiding blue prints.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows: in chapter 2, we present the problem statement, research objectives and contributions. We discuss the related work in chapter 3. Chapter 4

addresses error propagation probabilities and components reliability-based risk assessment. In Chapter 5, we generalize the reliability-based risk assessment to accommodate functional dependencies of the system. Chapter 6 deals with architectural metrics that captures the characteristics of the phenomenon of change propagation in software systems. We present maintainability-based risk concept and its estimation procedure and apply it in different maintenance contexts in chapter 7. In chapter8, an overview for the Software Architecture Risk Assessment (SARA) tool is described. Finally, we conclude and discuss future work in chapter 9.

2 The Problem

2.1 Problem Statement

Non-functional validation of software systems yet today does not find an appropriate consideration in the practice of software developers. Too little time and effort are devoted to this aspect during the software development process and a "fix-it-later" approach is still dominant. This allows software products to obey to the "short time to market" law, but their quality, as the ability to meet non-functional requirements, suffers of continuous (and sometime unaffordable) product updates after delivery.

This lack of validation appears even more serious if we consider that the software development has been rapidly going, in the last few years, towards component-based system configurations. Availability of self-contained software components (either previously developed or acquired from other companies/teams) is changing the software development process. Large effort has been spent to study the implications of reuse on the functional aspects of software systems, whereas the consequences of replacing a component (even with a functionally equivalent one) on the non-functional software aspects need more investigation.

Among the risks associated with non-functional attributes, a large significance has been given to software risk in the safety-critical system domain. Wherever software controls systems whose failures may be dangerous for environment and/or human life (e.g., aircrafts, nuclear plants, etc.), the consequences of software failures are better to be considered from the very early phases of the lifecycle. However, quantification of software risk is also suitable in other domains (independently of an absolute risk level), in order to detect components and events that may typically put in trouble the software system and the environment where the system will be running.

The risk of a software product has always been considered as a combination of the likelihood of a failure and the severity of "damages" that the failure may produce. The sources of failures are usually software behavioral faults, intended as behaviors that do not meet functional requirements. We refer to this type of risk as reliability-based risk.

In pursue of our research effort, we identify the following problems:

- How can we assess risk associated with the non-functional attributes of reliability and maintainability of a system based on quantitative means rather qualitative ones that are used in the current risk assessment methodologies

- How to define a practical reliability-based risk assessment methodology that captures the dependencies between the system functionalities and among components.

2.2 Research Objectives

Model-based risk assessment should guide the management of software development and maintenance. Anticipating what might go wrong and managing potential risks should be integrated into the software development process. Model-based risk assessment is capable of pinpointing the risky components of the system and helping in the allocation of the available resources to mitigate these risks. We are concerned with reliability-based risk taking into consideration of use-case relationships. Furthermore, the process of building software systems is quite complex and the word *change* characterizes many of the encountered problems: change in environment, change in users' expectations, change in organizational environment, and change in software requirements. Prolonging the software system life is highly profitable for the developing organization. Integrating the system with other systems, reusing it in different situations, or simplifying the maintenance process might help in extending the system life. Unfortunately, there are no universally used techniques for doing so.

As the need for software systems expands, development methodologies and techniques to automate the production of software and facilitate its maintainability is needed. Researchers look for techniques that reduce maintenance cost and improve quality. Using exiting solutions of recurring problems and cutting off applications development from scratch is one of the effective improvements to software productivity and maintainability.

The objectives of this research are:

- To generalize the reliability-based risk assessment methodology to account for the relationships between use cases and profile of execution of the use cases of the system.
- To generalize the assumption of failure occurrence independence in the components of the system and to account for error propagation among the components of the system in the methodology.
- To apply the generalized methodology on a real complex industrial case study, rather than on a hypothetical case study or an example adopted from the literature, which proves its value of pursuing this approach in tackling the problem of use case relationships.
- To develop a methodology for assessing maintainability-based risk for the system components based on the characteristics of the component and its interactions with other components. Sepecifically:
    - To introduce and define maintainability-based risk assessment for software architecture.

  - To develop a general framework to accommodate different software maintenance types.
  - To apply the estimation methodology on case studies with different size from different domains considering different types of software maintenance.
- To automate the methodology and to provide the analyst with tool support.

2.3 Research Contribution

This dissertation introduces a new approach for model-based risk assessment methodology for evaluating software architectures. The results of the research conducted in this thesis are (as discussed later in details):

- We address architectural attributes which differ from code-level software attributes. Architectural attributes focus on the level of components and connectors. In Specifics:
  - We introduce and define architectural attributes such as change propagation probabilities and size of change between components of the architecture.
  - We derive formulas for estimating these metrics using architectural level information.
  - We estimate error propagation probabilities, change propagation probabilities and size of change for several case studies from different domains.
  - We conduct empirical experiments to assess the estimation of these architectural attributes
- We relax some of the assumptions of the reliability-based risk assessment methodology [Goseva-Popstojanova+2003].
  - We generalize the methodology to accommodate relationships that model extensions and commonality within the UML use case model and apply it a real industrial case study which is a large command and control system used in a mission-critical application.It should be emphasized that, to the best of our knowledge, this is the first work which addresses relationships between use cases in assessing reliability-based risk.
  - We generalize the methodology to capture the dependencies among the system components and apply it on number of case studies.
- We are also concerned with maintainability-based risk analysis which assesses how difficult it is to maintain the system in the future because of possible maintenance tasks.
  - We introduce and define maintainability-based risk assessment for software architecture.

- We propose a methodology for estimating the maintainability-based risk when considering different types of maintenance: corrective, adaptive or perfective.
- We apply the proposed methodology on several case studies considering alternative maintenance types.
- We use Non-Homogeneous Poisson Process to get the maintainability-based risk estimate as a function of time when introducing new features in the system and/or due to adaptive maintenance.
- We automate the steps for the estimation methodology.

3 Related Work

In this chapter, we discuss literature review for scenario based risk assessment, reliability-based risk assessment and maintainability-based risk assessment.

3.1 Software Architecture Analysis Method (SAAM) and Architecture Trade-off Analysis Method (ATAM)

Software Architecture Analysis Method (SAAM) is a first generation "scenario-based" architectural assessment method. It was developed at the Software Engineering Institute (SEI) at Carnegie-Mellon University (CMU). SAAM focuses on functionality and ease of change. It is simple and easy to apply. The focus on function and modifiability can mask other problems as it neglects other quality properties. SAAM lacks wide stakeholder involvement. [SEI 2005]

Architecture Trade-off Analysis Method (ATAM) is second generation "scenario-based" architectural assessment method. It also was developed at the SEI. ATAM focuses on tradeoffs made between different requirements and quality properties. ATAM has two phases; the first phase is performed by architects, then stakeholders join in the second phase. [SEI 2005]

Risk assessment is conducted in both software evaluation approaches. In both approaches, the assessment is based on qualitative measures and the experience of the analyst.

3.2 Reliability-Based Risk Assessment

We developed a risk assessment methodology to be used in the early phases of the software life cycle [Goseva-Popstojanova+2003]. We used the Unified Modeling Language (UML) [OMG 2001] and commercial modeling environment Rational Rose Real Time (RoseRT) [Rational Rose RT] to obtain UML model statistics. First, a heuristic risk factor was obtained for each component and connector in software architecture. This factor combined severity and complexity (coupling) metrics for the components (connectors). Then, a Markov model was constructed to obtain scenarios' risk factors. The risk factors of use cases were estimated by averaging the scenarios' risk factors. This estimation methodology assumed independent use cases. Then, the overall system risk factor was estimated by weighting the independent use cases risk factors with the probability of their execution. We further identified critical components and connectors that would require careful analysis, design, implementation, and more testing effort.

Several other papers involved UML use cases in the analysis but they considered only independent use cases. In particular, in [Singh+ 2001] Singh et al. proposed an approach for reliability analysis of

component-based systems based on UML. In this work authors assumed independent use cases. Houmb et al. presented the CORAS UML profile for risk assessment in [Houmb+ 2002] and demonstrated its use on an e-commerce system assuming independent use cases. In [Hawkins+ 2002] Hawkins et al. addressed the hazard and safety analysis of object-oriented systems. Although the approach proposed in [Hawkins+ 2002] starts from the use cases, relationships among them were not considered.

3.2.1 Error Propagation Probabilities

To generalize the assumption of the independence of error occurrence in the components of the system, we used error propagation probabilities among the components of the system [Abdelmoez+ 2004A]. In this section, we first introduce and discuss the feature of *error propagation* in an architecture. Then, we review a derivative of this feature. Finally, we discuss related work to error propagation.

3.2.1.1 Error Propagation: Definition

We consider two components, say A and B, of an architecture, and we let X be the connector that carries information from A to B; for the purposes of our current discussion, the specific form of connector X is not important, we will merely model it as a set (of values that A may transmit to B). Also, the specific form of components A and B is not important; we will merely model them as functions that map an internal state and an input stimulus into a new state and an output.

Definition 1. The *Error Propagation Probability* from component A to component B is denoted by EP(A,B) and defined by:

$$EP(A,B) = Prob([B](x) \neq [B](x') \mid x \neq x'), \qquad (3.1)$$

where [B] denotes the function of component B, and x is an element of the connector X from A to B. We interpret [B] to capture all the effects of executing component B, including the effect on the state of B and the effect on any outputs produced by B.

We interpret EP(A,B) as the probability that an error in A is propagated by B (as opposed to being masked by B) because the outcome of executing B will be affected by the error in A. By extension of this definition, we let EP(A,A) be equal to 1, which is the probability that an error in A causes an error in A. Given an architecture with N components, we let EP be an N×N matrix such that the entry at row A and column B be the error propagation probability from A to B.

3.2.1.2 Unconditional Error Propagation

Note that the definition of the error propagation given above uses the concept of *conditional* probability, i.e. we calculate the probability that an error propagates from *A* to *B under the condition that A actually transmits a message to B*. It is often useful, however, to use the *unconditional error propagation* which we will denote simply as E(*A*,*B*), and define as the probability that an error propagates from *A* to *B* not conditioned upon the event that *A* sends a message to *B*. Function *E(A,B)* is clearly dependent on *EP(A,B),* but it further integrates the probability that *A* does send a message to *B*.

In order to bridge the gap between the original *(conditional) error propagation* and the newly introduced *unconditional error propagation*, let us consider the *transmission probability matrix* $\boldsymbol{T}$ where the entry $\boldsymbol{T}(A, B)$ reflects the probability with which the connector $(A \rightarrow B)$ gets activated during a typical/ canonical execution. $\boldsymbol{T}$ is the *NxN* matrix whose entry $\boldsymbol{T}(A, B)$ is the probability that the component *A* sends a message to component *B* given that the *A* is expected to transmit a message to *some* component. Note that:

- It is reasonable to assume that $\boldsymbol{T}(A, A) = 0$ for all components *A*,
- Clearly, $\boldsymbol{T}$ is a *stochastic* matrix, i.e. $\sum_{B} \boldsymbol{T}(A,B) = 1$ for every component *A*.

The matrix $\boldsymbol{T}$ is used to distinguish between a connector that is invoked intensively in each execution and one that is invoked only occasionally, under exceptional circumstances. The matrix $\boldsymbol{T}$ reflects the variance in frequency of activations of different connectors during a typical execution.

By virtue of simple probabilistic identities, we find that the *unconditional error propagation* is obtained as the product of the conditional error propagation probability with the probability that the connector over which the error propagates is activated, i.e.

$$E(A,B) = EP(A,B) \times \mathbf{T}(A, B). \qquad (3.2)$$

3.2.1.3 Related Work

In [Voas 1997], Voas analyzed error propagations between COTS components and presented an automated tool to simulate error propagation, which is used to deploy a fault injection experiment. This approach is code based and do not address the architecture of the system. Also, it is effort and time consuming even though it is supported with automated tool but it requires instrumentation for the code and then further analysis for the generated logs.

Michael et al [Michael+1997] presented an empirical study of data state error propagation behavior. The authors argue that at a given location either all data state errors injected tend to propagate to the

output, or else none of them does. This kind of analysis does not take the specific topology of the systems under consideration. It treats the system using their average behavior which we try to overcome by using error propagation probabilities.

In [Hiller+ 2001] Hiller et al. analyzed error propagation conceptually, introducing the concept of error permeability and discussing means to measure it using fault injection techniques. This work is similar to the error propagation study that we conduct but they consider how the error propagates through the component as an input/output relationship.

3.2.2 Estimating Error Propagation Probabilities

In [Abdelmoez+ 2004A], the authors have found that analytically, the error propagation probability, can be expressed in terms of the probabilities of the individual A-to-B messages and states, via the following formula:

$$EP(A \rightarrow B) = \frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A \rightarrow B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2} \qquad (3.3)$$

where $F_x^{-1}(y) = \{ v \in V_{A \rightarrow B} \mid F_x(v) = y \}$, and we assume a probability distribution P_B on the set of states S_B of component B, and a probability distribution $P_{A \rightarrow B}$ on the set of messages $V_{A \rightarrow B}$ passed from A to B. For further details, see Appendix II.

If we assume that the states of B, as well the messages passing through the connector from A to B are equi-probable, then the formula (3.3) for error propagation is simplified into

$$EP(A \rightarrow B) = \frac{1 - \frac{1}{|S_B||V_{A \rightarrow B}|^2} \sum_{x \in S_B} \sum_{y \in S_B} |F_x^{-1}(y)|^2}{1 - \frac{1}{|V_{A \rightarrow B}|}} \qquad (3.4)$$

In [Popic+ 2005], Popic et. al. extended their Bayesian reliability prediction of component based systems by introducing the error propagation probability. They studied the impact of the error propagation in a case study of an automated Personnel Access Control System. They concluded that error propagation has a significant impact on the system reliability prediction and, therefore, future architecture-based models should not ignore it.

3.2.3 Empirical Error Propagation

In order to validate our analytical study, we developed a framework for experimental error propagation analysis in which we utilize fault injection experiments to alter architecture specifications. We then simulated the corrupted specifications and record component traces as "faulty-run" logs. Finally we compared the faulty-run logs against a fault-free "golden-run" log obtained by simulating the uncorrupted architecture specifications. The framework is shown in Figure 1. We performed the simulation-based error propagation analysis in two phases: an acquisition phase and an analysis phase.

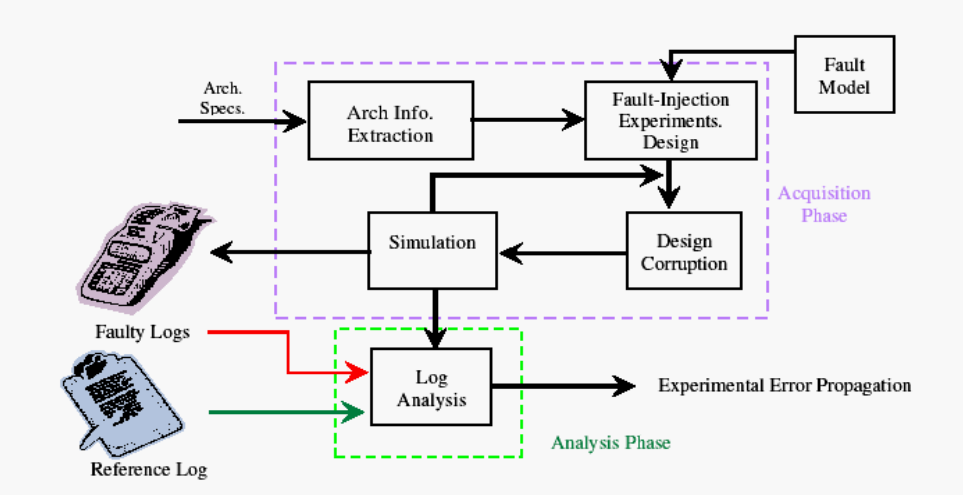


Figure 1 The framework of experimental error propagation analysis.

In the acquisition phase:

We extracted architecture information about the components and connectors that make up the software system. We used a message swapping fault model [Ammar+ 2001] to generate fault injection experiments. In each of the fault injection experiments we replaced all occurrences of a message nominally flowing over a connector (from component A to component B) by a different message (as a result of an error in component A) that belongs to the set of messages that A may send to B. We simulated the corrupted specifications and recorded simulation traces for the different experiments that cover all messages for the different connectors present in the architecture.

In the analysis phase:

We conducted post-simulation comparison between the faulty-run logs and the reference (fault-free) log. The comparisons were based on state transitions at simulation time instances following a fault injection. Immediately before injection of a fault, there were no difference between the state of component B as recorded on the reference log and its state recorded on the faulty log. After a fault was injected, any discrepancy between the two logs was due to error propagation from A to B. A faulty-message

propagating (from A to B) would at most cause a single instance of error propagation (from A to B). We computed the experimental error propagation probability from component A to B as the ratio of the fault injections (corresponding to errors in A) that propagated to component B over the total number of faults injected from component A to component B.

3.3 Maintainability-Based Risk Assessment

The cost of software maintenance accounts for 60% - 80% of the overall software system cost and enhancements (perfective/ adaptive) account for 78%-83% of the maintenance effort [Pigoski 1996]. Good maintainability of the system facilitates easy modifications when adapting to changes in the environment. Several studies address the quantification of hardware maintainability but only few attempt to quantify software maintainability. In [Oman+ 1994], Oman et al. presented the Maintainability Index (MI), which is a polynomial of widely used code level metrics. In [Muthanna+ 2000], Muthanna et al. conducted a similar study on the design level metrics to statistically estimate the maintainability of software systems.

The maintenance process introduces many types of risk such as project risk, usability risk and maintainability risk [Sherer 1997]. Maintainability-based risk assessment should guide the management of the software maintenance process. It can help maintainers in identifying the risky areas of the system. Hence, project managers can assign experienced maintainers to the risky areas.

Risk assessment can help with key decisions regarding allocation of resources. In [Papapanagiotakis+ 1994], the authors presented a model to allocate personnel and software/hardware environments utilities to maintenance efforts. This model assigned personnel based upon complexity indices for the application software, allowing for a threshold for skills and abilities of the servers assigned to different tasks. They suggested consideration of the level of software failure risk when allocating resources so that high failure risk modules requiring change were assigned to more experienced maintainers.

In this dissertation, we are introducing a general framework for assessing maintainability-based risk for system components. This framework assesses how difficult it will be to maintain the system in the future as a result of performing current maintenance tasks. It strives to capture the nature of the propagation of changes among the components when performing software maintenance. In the following subsections, we present related work for change propagation and change impact in software system, metrics to assess system maintainability and software maintainability models.

3.3.1 Change Propagation

Research of Change impact analysis was summarized in [Bohner+ 1996]. It dealt with finding all classes impacted by change. In the computer-aided mechanical design field, engineers used tools, such as

C-FAR system [Cohen+ 2000], to trace and predict change propagation. They divided the system under investigation into parts that are described according to their attributes. Interactions of the attributes were stated in semi C-FAR matrices. Then, the interactions were analyzed to predict the mutual effect between the attributes. C-FAR's computational complexity made it appropriate for small or relatively simple products.

Also, Design Structure Matrices (DSMs) [Steward 1981] estimated how change would propagate in a system. They are well-established techniques used to identify relationships between the components of the system or the design tasks [Eppinger+ 1994]. High connectivity found between design tasks suggested that high levels of dependency existed between the resulting system components. No indication such as the probability or scale of any such redesign was given by the DSMs.

In [Clarkson+ 2001], Clarkson et al. were concerned with the prediction and management of changes to an existing product resulting from faults or new requirements. They developed mathematical models to predict the risk of change propagation in terms of likelihood and impact of change.

In [Briand+ 1999B], Briand et al. investigated a probabilistic decision models based on coupling measurement to support impact analysis. They provided an ordering of classes where ripple effects were more likely. They investigated a commercial C++ system and they identified the coupling dimensions related to ripple effects and ranked the classes according to their probability of containing ripple effects. In [Briand+ 2003], Briand et al. proposed a UML model-based approach to impact analysis that can allow early decision-making and a change planning process. They first made a consistency check for UML diagrams. Then they identified changes between two different versions of a UML model according to a change taxonomy, and determined model elements that are impacted by changes using defined impact analysis rules. They prioritized the results of impact analysis according to the likelihood of occurrence using a measure of distance between a changed element and potentially impacted elements. They also presented a prototype tool that provides automated support for their impact analysis strategy.

In [Fanta+ 1998], Rajlich et al studied refactoring of an object oriented C++ code which had a number of misplaced functions. They developed some tools to reengineer the code. In [Rajlich+ 2000], they summarized the evolution role in the lifecycle of a software. Software lifecycle was split into stages and the characteristics of each stage were highlighted. Change propagation in evolutionary development had been the focus of prior models [Rajlich 2000], [Schach+ 2000]. In these models, a program was divided into parts that were used to construct a propagation graph. These Techniques focused upon the potential necessary changes that were required in redesigning the software. They used program variables to locate links that might propagate the change, but they only predicted one step of change at a time.

In [Rajlich+ 2002], Rajlich et al. presented a technique for unanticipated incremental change and a case study. The technique emphasized the role of software comprehension and the role of programming concepts. In [Rajlich+ 2004], Rajlich et al. presented a set of selected incremental change activities—change request, concept extraction, concept location, impact analysis, actualization, incorporation, change propagation, refactoring, and role splitting—in which programming concepts and program dependencies played a key role.

In [Hassan+ 2004] , Hassan et al. suggested a number of heuristics to anticipate change propagation in software systems. In order to evaluate the performance of the proposed heuristics, they presented a measuring framework to calculate recall and precision. They analyzed the history of development for five large open source software systems to validate empirically their results. Their results doubted the efficiency of code structure heuristics as indicators for change propagation. Furthermore, they concluded that the historical change data could be used to develop better heuristics to assist developers during the change propagation process.

In [Tsantalis+ 2005], The authors proposed a probabilistic approach to estimate the change proneness of an object-oriented design. They evaluated the probability that each class of the system will be affected when adding new functionality or when modifying existing functionality. Their proposed model had been evaluated on two multi-version open source projects.

3.3.2 Maintainability Metrics

In this subsection, we review the literature for metrics and models used to assess the fuzzy maintainability concept of software systems. Early work of maintainability focused on source code metrics. Later, maintainability models tried to evaluate it using a set of metrics rather that relying on a single metric.

3.3.2.1 Traditional Metrics

In [Rombach 1987], Rombach advocated that source code metrics can predict maintenance effort, maintainability, comprehensibility, locality and modifiability. Also in [Kafura+ 1987], Kafura et al. found correlation between software code metrics and developers perceived maintainability.

In [Stark+ 1994], Stark et al. proposed a set of software metrics to assist in managing corrective and adaptive maintenance processes. This set followed the Goal/Question/Metrics paradigm. Some of these metrics tried to maximize effort and schedule by answering the question “How maintainable is the system?”. Software size and software complexity metrics can assist answering that question Line of Code (LOC) is the mostly used metric for measuring software size. Halstead metrics is the first metrics that try to capture the size by other means rather than counting the lines of code [Halstead 1977]. McCabe’s

Cyclomatic Complexity is used usually as a measure for software complexity. It measures the number of independent execution paths in the software [McCabe 1976].

A complexity measure should be useful in predicting maintenance costs. In [Gill+ 1991], authors showed how the cyclomatic complexity metric relates to software maintenance productivity. They introduced a simple transformation to the cyclomatic complexity metric, dividing it by the size of the system, resulting into a complexity density ratio. This complexity density ratio showed to be useful in predicting software maintenance productivity on small maintenance projects.

The following is brief overview of these traditioal metrics :

- **Line of Code (LOC)** is one of the mostly used metric. This is because of LOC is the most available size metrics, as most of the editor programs count the lines of the file being edited. There are different kinds of line of code such as blank lines, comment lines, lines with more than one instruction and program headers. According to [Grady+ 1987], non-commented Lines of code **NLOC** considers all lines that contains any program statement other than blank or comments.

- **Halstead metrics** are one of the first metrics that try to capture the size by other means rather than counting the lines of code [Halstead 1977]. They try to capture the physical and psychological aspect of the software. They are based on the following:

  μ_1 = number of unique operators. μ_2 = number of unique operands.

  N_1 = total occurrences of operators. N_2 = total occurrences of operands.

  From these basic measurements, Halstead derived different metrics such as program vocabulary (n), length (N), volume (V), total effort (E) and development time (T). These derived metrics received lots of criticism such as in [Card+ 1990] and [Fenton+ 1996] because they lack any theoretical or empirical foundations.

- **McCabe's Cyclomatic Complexity** measures the number of independent execution paths in the software program. In [McCabe 1976], McCabe suggested that small number of cyclomatic complexity increases the module testability and understandability. He calculated the cyclomatic complexity as:

$$V(G) = e - n + 2 \tag{3.5}$$

  Where

  $V(G)$ = cyclomatic complexity of graph G.

  e = number of edges.

  n = number of nodes.

In [Grady 1994], the author analyzed 830KLOC FORTRAN code and found a strong relation between cyclomatic complexity and number of module updates. He suggested that no module cyclomatic complexity should exceed 15. On the hand, [Fenton+ 2000] showed that the cyclomatic complexity does not correlate with fault metrics but when combined with other metrics some correlation was found. Furthermore in [Card+ 1990], authors found weak correlation between cyclomatic complexity and fault metrics.

- **The Chidamber and Kemerer object oriented metrics**: [Chidamber+ 1994]
    - Coupling Between Objects (CBO) is defined as the number of the components coupled to a component in an architecture.
    - Response For a Class (RFC) is the sum of all the methods that can be invoked in response to a message to an object of a component or by a method of the component.
    - Weighted Methods per Class (WMC) is the sum of the complexities of all the methods in a component.
- **Message Passing Coupling** (MPC) is the sum of the number of method calls made by all the methods in a component. [Briand+ 1999A]

3.3.2.2 Maintainability Metrics Model

There have been several studies trying to characterize and quantify software maintainability. One of the famous studies by Oman et al. introduced the Maintainability Index measure. In [Oman+ 1992], he developed the MI equations. The study indicated that widely-used measures; such as Halstead measures and McCabe's cyclomatic complexity; are good predictors of maintainability. In [Oman+ 1994], Oman introduced a modification of the MI and described how to calibrate it using large suite of industrial-use operational code. Oman developed a prototype tool to support capture and use of maintainability measures for Pascal and C [Oman 1991]. MI is given by a polynomial in the following form

$$MI = 171 - 5.2 * \ln(aveV) - 0.23 * aveV(g') - 16.2 * \ln(aveLOC) + 50 * \sin(\sqrt{2.4 * perCM}) \quad (3.6)$$

The terms are defined as follows:

aveV = average Halstead Volume V per module.

aveV(g') = average extended cyclomatic complexity per module.

aveLOC = the average lines of code (LOC) per module; and

perCM = average percent of lines of comments per module

In a joint research effort, the software maintainability metrics models were used to quantify maintainability via Maintainability Index (MI) [Welker+ 1995]. Measurement and use of the MI was integrated as part of the overall development or maintenance process. These efforts indicated that MI measurement applied during software development could help reduce lifecycle costs. It worth noting that the maintainability-index lack theoretical foundations and was conducted on rather small systems with current consideration. Even though, it is one of the most reported studies and there is no further investigations on how helpful it is or how widely it is used. In contrast, our proposed methodology is based on architectural artifacts so it can be applied early on in the life cycle where maintainability index relies on code-level metrics. Moreover, our methodology tries to capture the change propagation phenomena that affect the maintenance process. Also, our methodology is generic enough to consider different types of maintenance: corrective, adaptive and perfective.

In [Muthanna+ 2002], Muthanna et al. conducted a similar study but rather on the design level metrics to statistically estimate the maintainability of software systems. They constructed a linear model based on a minimal set of design level software metrics to predict Software Maintainability Index (SMI), as follow:

$$SMI=125–3.989\cdot FAN_{avg}–0.954\cdot DF–1.123\cdot MC_{avg} \quad (3.7)$$

Where FAN_{avg} is the average number of external calls coming from this module.

DF is the number of incoming and outgoing dataflow for the module.

MC_{avg} is the average cyclomatic complexity for the module.

The authors only validated the model on a single 92KLOC industrial software system against developers' opinions. According to the authors, the model gave a good prediction in most cases. There is a restriction to use this model; the system should be decomposed into modules of 1 to 2KLOC. Also, this work lacks any theoretical foundation. On the other hand, our methodology tries to capture the change propagation phenomena that run behind the scene. Also, our methodology is accommodate different types of maintenance: corrective, adaptive and perfective while this work does not differentiate between them.

In [Menzies+ 2000], the authors tried to assess the maintainability of software systems. They suggested a theoretical model of maintenance effort. They assumed that a large portion of maintenance cost is spent on continual retesting. They depicted testing by the pathways that reach from inputs to some interesting zone of a program, i.e. a bug or a desired feature. Their goal of testing was to show that a test set uncovers no bugs while reaching all desired features. If the system was hard to test; i.e. if it had low reachability, it was hard to maintain. Therefore, easy maintenance required easy testing and easy testing required easy reachability. They developed and simulated a model of system reachability. Then, they used

a sensitivity analysis to find the key parameters that change system reachability. This model assumes an average topology for the system and does not into consideration the specifics of the architecture topology of a certain system. Also, it concentrates only on corrective maintenance and fixing bugs. While, our proposed methodology takes into account other maintenance types beside corrective maintenance and takes into consideration the specifics characteristics of the components of the system.

3.3.3 Request Generation Using Non-Homogeneous Poisson Process

In [Burch+ 1997] and [Gefen+ 1996], the authors observed the non-homogeneous nature of maintenance request arrivals. They found an empirical support for the evolution of the mean rate of maintenance requests overtime. In [Tan+ 2005], Tan et al. modeled the random arrival of maintenance requests using non-homogeneous Poisson process with time varying mean rate. In their model two sources with time varying mean rate. In their model two sources of system enhancements were considered:

- Adaptive maintenance: They assumed that the mean rate of request arrivals of adaptive maintenance can be constant or increasing with time
- Perfective maintenance: They assumed that the mean request rate M.g(z) of perfective maintenance for a certain feature can be modeled as inverted U-shape starting from the time of introducing this feature to the system [Burch+ 1997].

Thus, the total mean arrival rate of requests, $\lambda(t)$, is the sum of the mean arrival rates from theses two sources. $\lambda(t)$ can be given by the following:

$$\lambda_i(t) = h(t + \sum_{l=1}^{i-1} T_l) + \sum_{k=0}^{i-2} M_k \cdot g(t + \sum_{l=k+1}^{i-1} T_l - L_{k+1}) + M_{i-1} \cdot g(t - L_i) \cdot \theta(t - L_i) \qquad (3.8)$$

Where:

- t is the elapsed time since the introduction of the system
- T_i denote the interval between the $(i-1)^{th}$ and the i^{th} maintenance activity. Assuming that, in the system useful lifetime.T, there are r maintenance activities.
- M_0 is starting function points of the system and M_i function points are added during the i^{th} activity.
- L_i is the time to implement M_i function points (L_1 =0), and
- q(.) is the step function.
- H(t) mean rate of request arrivals of adaptive maintenance

- M.g(z) is the mean rate for request arrivals of perfective maintenance of a new system feature of complexity M. g(z)is given by:

$$g(z) = K \frac{\beta^{\alpha+1}}{\Gamma(\alpha+1)} z^{\alpha} \exp(-\beta z) \qquad (3.9)$$

where

- α and β are factors reflecting the learning effect and the saturation effect characteristics of the users of the system.
- The term $\Gamma(\alpha+1)$ is the Gamma function.
- z is the time measured from the first introduction of the new feature to the users

4 Error Propagation Probabilities and Reliability-Based Risk Assessment

In this chapter, we address error propagation probabilities as an architectural attributes. Also, we deal with the generalization for the assumption of error occurrence independence in the components of the system. We account for error propagation among the components of the system. Furthermore, we use error propagation probabilities in refining the reliability-based risk assessment methodology.

4.1 Analytical Error Propagation Results

As a case study, we inspect a command and control system used in a mission-critical application. We present only the analysis of the Internal Thermal Control subsystem. The detailed artifacts of the command and control case study are in Appendix I.A. Using the architecture of a command and control system, we first estimate the error propagation probabilities analytically. We get the set of states S_B and messages $V_{A\rightarrow B}$ from the artifacts of the system specification. We obtain the matrix EP of (conditional) error propagation probabilities of this system, using the equation (3.3). The analytical error propagation probabilities of the case study are shown in Table 1.

Table 1 Conditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.1061	0.4210	0.3368	0.4472	0.4623				
	C2	0.2001							0.5238		
	C3	0.0105	0.4722								
	C4	0.0190	0.2332								
	C5		0.2765								
	C6		0.1265								
	C7	0.3761									
	C8										
	C9										
	C10	0.0014									

To illustrate how we compute the (conditional) error propagation, we work out our computation steps for *EP(2→8),* the (conditional) error propagation from component 2 to component 8:

- From Figure 2 , the state diagram of component 8, we determine the set of states S_8={*s1, s2*}. From Figure 3, the message protocol between component 2 to component 8, we determine set of messages $V_{2\rightarrow 8}$={*m1, m2, m3, m4*}.

- From simulating the operational profile, we estimate the probability distribution of states S_B of component 8 and the probability distribution of messages $V_{A\to B}$ exchanged between component 2 and component 8. These are given in Table 2 and Table 3, respectively.
- Considering Figure 2 for the state diagram of component 8, we get $F_x^{-1}(y) = \{v \in V_{A\to B} \mid F_x(v) = y\}$ for each pair of states in component 8. For example, we find that $F_{s1}^{-1}(s2) = \{m2\}$, which means that the message m2 is the set of messages that cause us to make a transition to state s2 given that we are in state s1. Then, the term $P_{2\to 8}\,[F_{s1}^{-1}(s2)]^2$ will be $P_{2\to 8}\,[\{m2\}]^2$ that we calculate using Table 3.
- Similarly, we get $F_{s2}^{-1}(s2) = \{m3, m4\}$ and $P_{2\to 8}\,[F_{S2}^{-1}(S2)]^2$ will be $P_{2\to 8}\,[\{m3,m4\}]^2$. Note that we need to consider all the messages for a certain state. For state S2, the message m1 is not defined from the state transition diagram. So, we assume that receiving a message m1 given that we are at state s2 will cause us to remain in State s2 (Self reflective transition). As a result, we get $F_{s2}^{-1}(s2) = \{m1, m3, m4\}$ and $P_{2\to 8}\,[F_{S2}^{-1}(S2)]^2$ will be $P_{2\to 8}\,[\{m1,m3,m4\}]^2$. Then the state transition diagram of component 8 will be as in Figure 4 to take undefined messages into consideration.
- According to equation (3.3), we continue evaluating these probabilities for each pair of states in component 8, multiply them with the corresponding state probabilities P_B and use the probabilities of the messages $V_{A\to B}$ between component 2 and component 8. Then, we get $EP(2\to 8)=0.5238$.

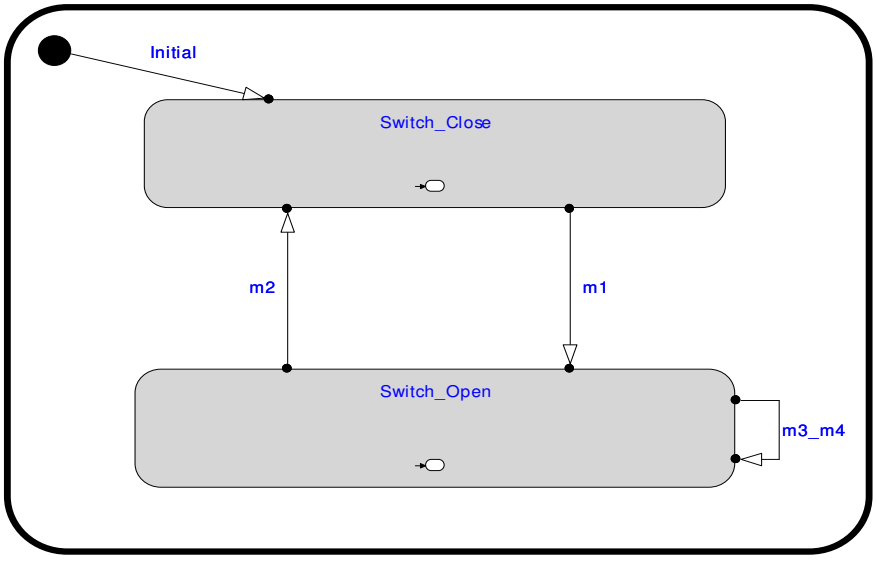


Figure 2 A state diagram of component 8.

Table 2 Probability distribution of states S_B of C 8

State	Switch Open (S1)	Switch Closed (S2)
Prob(State)	0.99	0.01

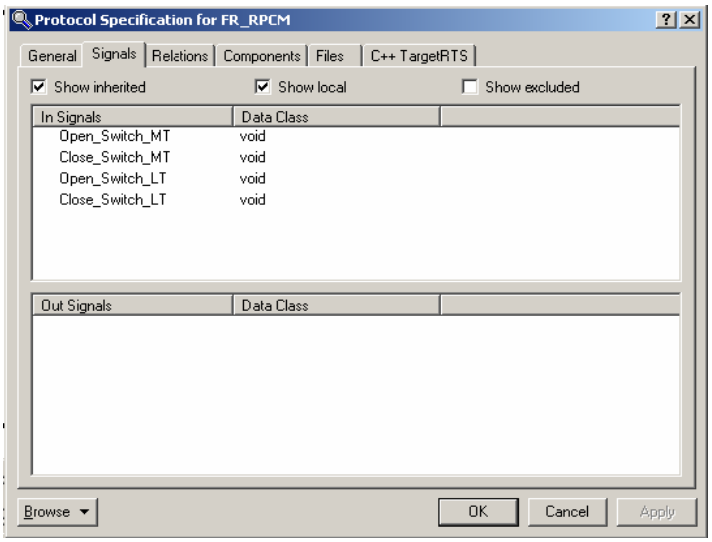


Figure 3 A sample of a sanitized message protocol (components 2 and 8).

Table 3 Probability distribution of messages $V_{A\rightarrow B}$ exchanged between C 2 and C 8

Message	m1	m2	m3	m4
Prob(Message)	0.267	0.267	0.266	0.2

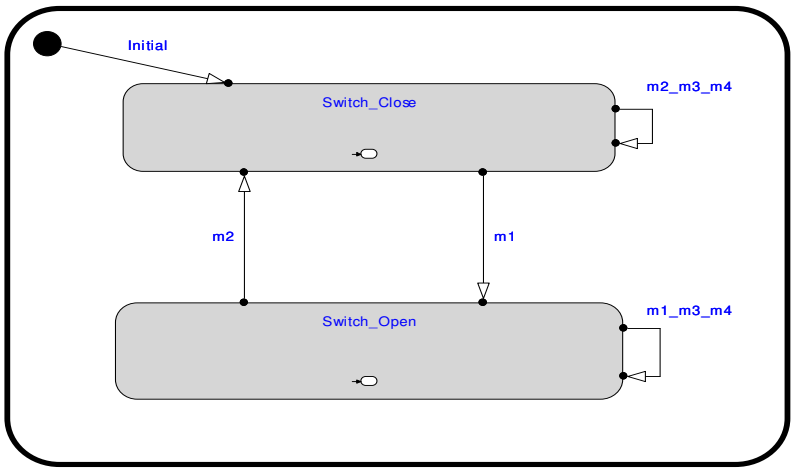


Figure 4 Updated state diagram of component 8.

4. Error Propagation Probabilities and Reliability-Based Risk Assessment

For this particular case study, we have derived the connector activation matrix ***T*** as a stochastic matrix of probabilities that contains for each entry (A,B), the probability that connector (A,B) is activated, given that component A is broadcasting a message**.** Using this connector activation matrix, we derive the *unconditional error propagation matrix* E_A**,** also referred to as the 1-step error propagation matrix of the system; this is given in Table 4. We get the matrix ***T*** through the simulation of the system representing the operational profile of the execution.

Table 4 Unconditional Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.0012	0.0132	0.0102	0.0146	0.0145				
	C2	0.1104							0.1264		
	C3	0.0060	0.2024								
	C4	0.0107	0.1026								
	C5		0.1005								
	C6		0.0506								
	C7	0.3761									
	C8										
	C9										
	C10	0.0014									

Using the unconditional error propagation matrix, say E_A, given above, we derive the matrix of cumulative error propagation probabilities, which we call E^*_A. Table 5 gives cumulative error propagation probabilities matrix for the command and control case study. Except for possible round-off errors, matrix E^*_A is greater than matrix E_A, entry by entry.

Table 5 Cumulative Error Propagation Matrix - Analytical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.00	1.16E-03	1.32E-02	1.02E-02	1.46E-02	1.45E-02		1.46E-04		
	C2	1.10E-01	1.00	1.46E-03	1.12E-03	1.61E-03	1.60E-03		1.26E-01		
	C3	6.04E-03	2.02E-01	1.00	6.15E-05	8.82E-05	8.78E-05		2.56E-02		
	C4	1.07E-02	1.03E-01	1.41E-04	1.00	1.56E-04	1.55E-04		1.30E-02		
	C5	1.11E-02	1.01E-01	1.47E-04	1.13E-04	1.00	1.61E-04		1.27E-02		
	C6	5.59E-03	5.06E-02	7.39E-05	5.69E-05	8.15E-05	1.00		6.40E-03		
	C7	3.76E-01	4.35E-04	4.98E-03	3.83E-03	5.49E-03	5.47E-03	1.00	5.49E-05		
	C8								1.00		
	C9									1.00	
	C10	1.41E-03	1.62E-06	1.86E-05	1.43E-05	2.05E-05	2.04E-05	0.00	2.05E-07	0.00	1.00

4.2 Experimental Error Propagation Results

Table 6 shows the experimentally obtained error propagation matrix E_E from the fault injection experiment explained in section 3.2.3 considering the same mode of operation whose analytical error propagation matrix E_A is given in Table 4.

Table 6 Unconditional Error Propagation Matrix E_E - Empirical Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1		0.5000	0.0557	0.4912	0.1331	0.1280				
	C2	0.7838							0.4286		
	C3	0.0161	0.7083								
	C4	0.7917	0.6429								
	C5		1.0000								
	C6		1.0000								
	C7	0.7500									
	C8										
	C9										
	C10	1.0000									

Using matrix E_E, we derive matrix E^*_E of cumulative error propagation probabilities, and find the results shown in Table 7. Note that except for round-off errors, this matrix is greater than the matrix of unconditional probabilities, entry by entry.

Table 7 Cumulative Error Propagation - Experimental Results

		B									
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
A	C1	1.00	5.00E-01	5.57E-02	4.91E-01	1.33E-01	1.28E-01		2.14E-01		
	C2	7.84E-01	1.00	4.37E-02	3.85E-01	1.04E-01	1.00E-01		4.29E-01		
	C3	1.61E-02	7.08E-01	1.00	7.91E-03	2.14E-03	2.06E-03		3.04E-01		
	C4	7.92E-01	6.43E-01	4.41E-02	1.00	1.05E-01	1.01E-01		2.76E-01		
	C5	7.84E-01	1.00E+00	4.37E-02	3.85E-01	1.00	1.00E-01		4.29E-01		
	C6	7.84E-01	1.00E+00	4.37E-02	3.85E-01	1.04E-01	1.00		4.29E-01		
	C7	7.50E-01	3.75E-01	4.18E-02	3.68E-01	9.98E-02	9.60E-02	1.00	1.61E-01		
	C8								1.00		
	C9									1.00	
	C10	1.00E+00	5.00E-01	5.57E-02	4.91E-01	1.33E-01	1.28E-01		2.14E-01		1.00

4.3 Error Propagation Probabilities Validation

In this section, we confront the results computed by the analytical formula against the results derived from the fault injection experiment to assess the validity of our analytical formulas. We let E_A and E_E be (respectively) the analytical matrix and the empirical matrix of (unconditional) error propagation for our sample architecture; these are both 10×10 matrices. We use a number of criteria to this effect:

- The first possible criterion is simply the correlation between the entries of the two matrices; because these matrices contain 100 values each, the correlations do bear some significance.

- The second possible criterion is to correlate, not all the values of the matrices, but rather the non-trivial values (other than those that are either 0 or 1 by definition); the rationale behind this criterion is that trivial values do not really test our analytical results.

- The third criterion discriminates between empirical values that were derived from a small number of fault injections and those that were derived from a large number of fault injections. If our analytical results are accurate, we should find empirical values that stem from large numbers of fault injections to be highly correlated to their corresponding analytical values, whereas those values than stem from small numbers of fault injections are not guaranteed to correlate to their corresponding analytical results.

Orthogonally, we find it useful to compare not only E_A and E_E, which represent single step propagations, but also cumulative versions of these matrices, which represent probabilities of error propagations that may have taken more than one step through the architecture. There are two reasons why we may want to consider the cumulative matrix E* in addition to the single-step matrix E:

- In practice, if we are interested in the probability that an error in A propagates to B, we usually do not care in how many steps the propagation takes place; hence E* is a better reflection of what we want to measure than E.

- Also, if there is any discrepancy between what the analytical study defines as a single step and what the empirical study does, this discrepancy will be smoothed out once we consider propagations of arbitrary length.

In the remainder of this sub section, we apply these criteria to matrices E and E* in order to determine to what extent the values obtained empirically are consistent with those found analytically.

4.3.1 Correlating One Step Matrices

In this section, we present the results of the study that we conducted to explore the correlation between the analytically estimated single step error propagation matrix and its experimentally derived counterpart. The correlation coefficient between all the cells of the analytical E_A matrix and the experimental E_E matrix is:

$$Cor(E_A,E_E) = 0.628 \qquad \text{(r value)} \qquad (4.1)$$

where 'r' denotes the Pearson product-moment correlation coefficient.

We note, however, that there are only 15 non-trivial entries in each of the two matrices. Trivial entries correspond to self-loops from a component to itself (with error propagation probability of 1 by definition) and to the non-directly connected components (with error propagation probability of 0 by definition). It may be useful to evaluate the correlation between the set of non-trivial values of matrices E_A and E_E having a significant number of fault injections(>20). The connectors that has <20 fault injections are shaded in Table 8 We find:

$$Cor'(E_A,E_E) = 0.5576 \qquad \text{(r value)} \qquad (4.2)$$

Table 8 contains the 15 non-trivial entries corresponding to the 15 connectors over which faults were injected during the controlled experiment. Note that the number of injected faults over connectors varies considerably across the entries. The connectors in the table follow a descending order with respect to the number faults injected over each connector. Overall, the correlation decreases as the number of injected

faults drop, although not monotonically; the disturbances in the first few rows may stem from the fact that a correlation is not necessarily meaningful when too few values are involved.

The results in this table are interesting, in that they show a fairly high correlation between experimental results and analytical results in those cases where the experimental result is based on a large number of fault injections. Also, predictably, the correlation drops (as shown in Table 8) as the number of fault injections drop (though not monotonically).

Table 8 Correlation between analytical and experimental EP probabilities

Entry	Connector		E_A	E_E	# Injected Faults	Correlation Coefficient	
	From	To					
1	1	5	0.0146	0.1331	1067	N/A	N/A
2	1	6	0.0145	0.1280	1055	1.0000	Entries 1 through 2
3	1	3	0.0132	0.0557	592	0.9999	Entries 1 through 3
4	3	1	0.0060	0.0161	559	0.8708	Entries 1 through 4
5	7	1	0.3761	0.7500	64	0.9894	Entries 1 through 5
6	1	4	0.0102	0.4912	57	0.8160	Entries 1 through 6
7	2	1	0.1104	0.7838	37	0.7153	Entries 1 through 7
8	1	2	0.0012	0.5000	36	0.6488	Entries 1 through 8
9	4	2	0.1026	0.6429	28	0.6433	Entries 1 through 9
10	3	2	0.2024	0.7083	24	0.6829	Entries 1 through 10
11	4	1	0.0107	0.7917	24	0.5576	Entries 1 through 11
12	2	8	0.1264	0.4286	7	0.5501	Entries 1 through 12
13	5	2	0.1005	1.0000	4	0.5068	Entries 1 through 13
14	6	2	0.0506	1.0000	4	0.4291	Entries 1 through 14
15	10	1	0.0014	1.0000	4	0.3240	Entries 1 through 15

4.3.2 Correlating Cumulative Matrices

In addition to analyzing the correlation between matrices E_A and E_E (which represent the one-step unconditional error propagation probabilities, estimated analytically and experimentally), we are interested in analyzing the correlations between matrices E^*_A and E^*_E, which represent the cumulative (multi-step) versions of these matrices. The results of our study are shown in Figure 5. We find for all elements,

$$\text{Cor}\,(E^*_A, E^*_E) = 0.737 \qquad \text{(r value)} \qquad (4.3)$$

Also, we find that the correlation for multi-step error propagation for non-trivial values, is

$$Cor'(E^*_A, E^*_E) = 0.460 \qquad \text{(r value)} \qquad (4.4)$$

Hence our analytical formula can be used to predict cumulative error propagation probabilities throughout an architecture with a significant positive correlation, at least in this sample case study - all the while using nothing more than the UML-RT description of the system.

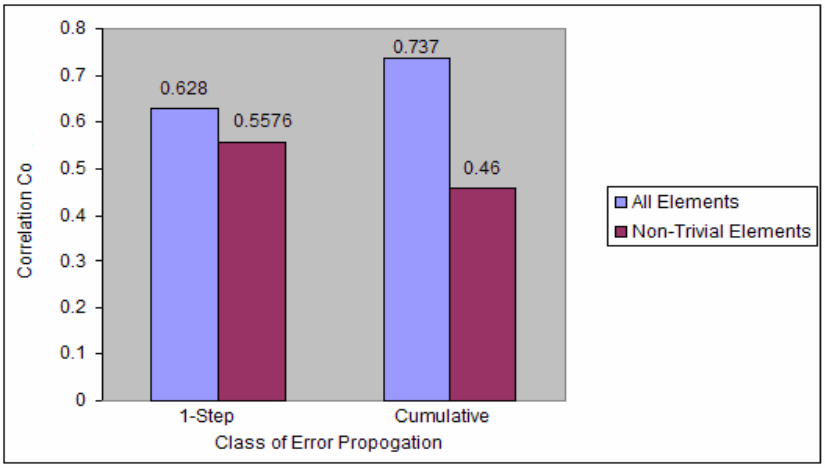


Figure 5 Correlation between analytical and empirical error propagation

4.3.3 Statistical Significance of the Correlations

Now we need to validate our results, i.e. to make sure that the positive correlation values we observed are statistically significant. To further test the relationship between analytical and experimental error propagation hypothesis testing was done using the T-test (One-tail) [Moore+ 2003] for the non-trivial entries using the level of significance $\alpha = 0.05$

- *H0* : $\rho = 0$ (There is no linear association between analytical error propagation values and empirical error propagation values)
- *H1* : $\rho > 0$ (There is a positive linear association between analytical error propagation values and empirical error propagation values)

The null-hypothesis is rejected when the p value is smaller than 0.05.

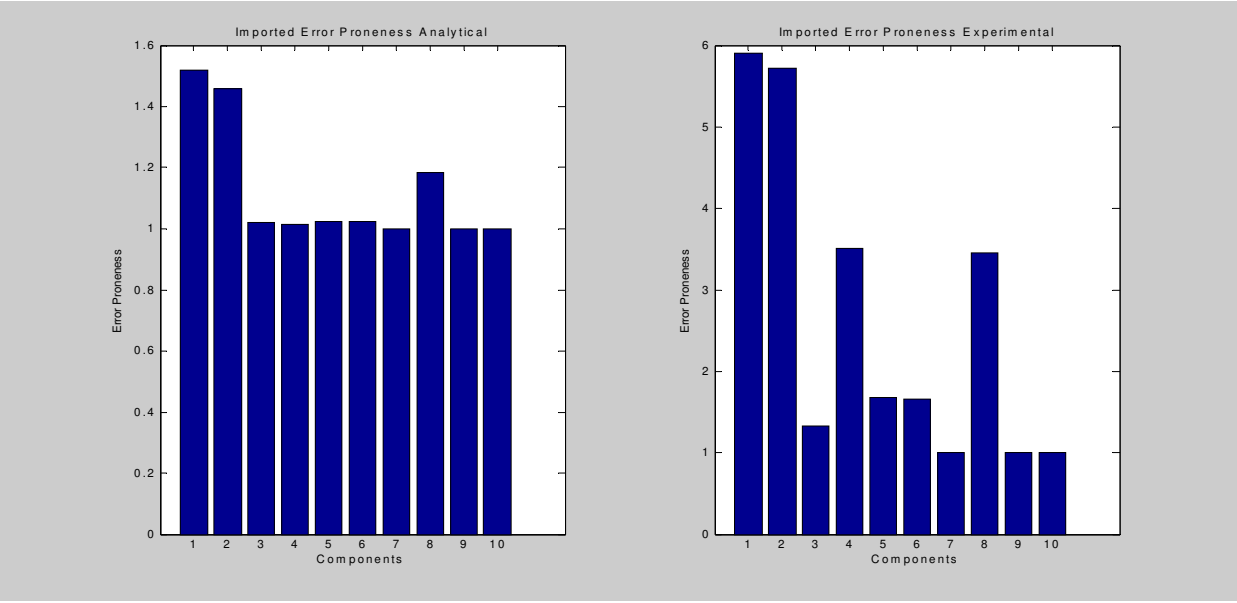
We have computed the value of t statistic for the non-trivial values of 1-step matrices $t_{ob} = 2.015$ (n=11),and the corresponding $P < 0.05$; whence we infer that the correlation of 0.628 is statistically

significant. Likewise, the value of t statistic for the non-trivial values of cumulative error propagation matrices was found to be $t_{ob} = 3.5893$(n=50), and the corresponding P < 0.05 which shows that the correlation between experimental and analytical cumulative error propagation matrices is statistically significant.

The T-test results showed that there is a linear association between analytical error propagation and experimental error propagation as well as between cumulative analytical error propagation and cumulative experimental error propagation as in both cases based on the P value the null hypothesis of no linear association was rejected.

4.4 Ranking Components According to their Error Proneness

One of the usages of the error propagation matrix is to check for components that tend to be affected by errors arising throughout the architecture; component error proneness. Thus, this calls for providing these components with fault tolerance capabilities (error detection, damage assessment, error recovery).



(a) Analytical imported error proneness (b) Experimental imported error proneness

Figure 6 Imported error proneness for command and control system case study

We use the column sum of the error propagation matrix as measure for the imported error proneness of the component then we ranked the components accordingly. In Figure 6, we show the results of the

component error proneness estimated from analytical and experimental error propagation matrices. If we compare the ranking of the component in both cases, we find that the components with highest error proneness identified from analytical results are the same as those identified from experimental results (components C1 and C2).

Furthermore, we want to study the behavior of error proneness of the components across the steps of propagation of the error. First, let us assume uniform initial error proneness for the components of the system. Then multiply the initial error proneness by the powers of the analytical unconditional error propagation matrix to get the 1-step, 2-steps… error proneness of the components, as shown in Figure 7. From the results, we find that components C1 and C2 are the most error prone across the steps of propagation of the error. Thus, they are the most critical components considering error proneness (similar to the results of the column sum).

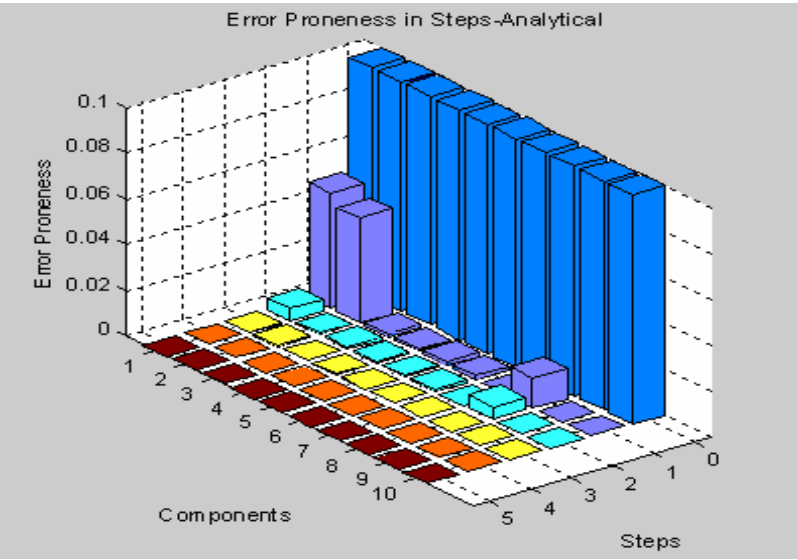


Figure 7 Analytical error proneness of the components in steps

4.5 Considering Error Propagation Probabilities in Assessing Components Reliability-Based Risk

In [Goseva-Popstojanova+2003], we estimate component's reliability-based risk factor as a product of the dynamic complexity and the severity level of that component for a certain scenario. (See Section 5.1.2 for details). To get an estimate for the system-level reliability-based risk factor for the component, we can use static cyclomatic complexity as a predictor for the probability of error occurrence in that component and multiply it by the system-level severity. Such estimation assumes independence of error occurrences in the components of the system. To generalize this assumption, we use error propagation probabilities

among the components of the system in reliability-based risk assessment. We use the static component complexity as measure of the probability of having initial failures. Then, we use the error propagation probabilities to get the unconditional error probability of the components to account for the effect of the dependency among the components of the system. Thus, the reliability-based risk factor of a system component C_i will be

$$\mathrm{rrf}_i = [\ \sum_j \mathrm{iep}_j * \mathrm{ep}_{ij}\ ] * \mathrm{csv}_i \qquad (4.5)$$

Where RRF = [rrf_i] is the Component Reliability-based Risk Factor

IEP = [iep_i] is the Initial Error Probability

EP = [ep_{ij}] is the the Error Propagation Probabilities matrix

CSv = [csv_i] is the Component Severity

It is worth noting that cyclomatic complexity is a good predictor for fault density. As many faults do not manifest themselves into errors, thus there is weak correlation between fault density and error density. Even though, in a risk assessment context, we consider worst-case scenario, so it is justifiable to use cyclomatic complexity as a first-order approximation for initial error probability in the early life-cycle of the system. Also, cyclomtic complexity is used also by NASA as on the attributes to assess error potential in their SILAP risk assessment process [Costello 2005].

In the following subsections, we estimate the reliability-based risk for the components of the system. First, we use the normalized cyclomatic complexity of the system components as an estimate for the initial error probabilities IEP. Then, we use the system artifacts to estimate the error propagation probabilities EP. Finally, the severity of the system components CSv is assessed according to MIL_STD_1629A [MIL_STD_1629A]. Severity considers the worst case consequence of a failure determined by the degree of injury, property damage, system damage, and mission loss that could ultimately occur. Based on hazard analysis, we identify the severity classes: *Catastrophic*, *Critical*, *Major* and *Minor*. The assignment of component severity level of each component is based on the hazard analysis conducted by domain experts knowledgeable about these case studies.

4.5.1 Pace Maker Case Study Results

A cardiac pacemaker [Douglass 1998] is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. An error in the software operation of the device can cause loss of a patient's life. The detail of this case study is introduced in Appendix I.B.

4. Error Propagation Probabilities and Reliability-Based Risk Assessment

The cyclomatic complexity of the pace maker components are shown in Figure 8. By normalizing the cyclomatic complexity by their total sum, we get as an estimate for the initial error probabilities IEP, given in Figure 9 . Using the pace maker artifacts, the pace maker error propagation probabilities results are estimated, shown in Figure 10. Finally, the severity of the system components CSv is assessed and shown in Table 9.

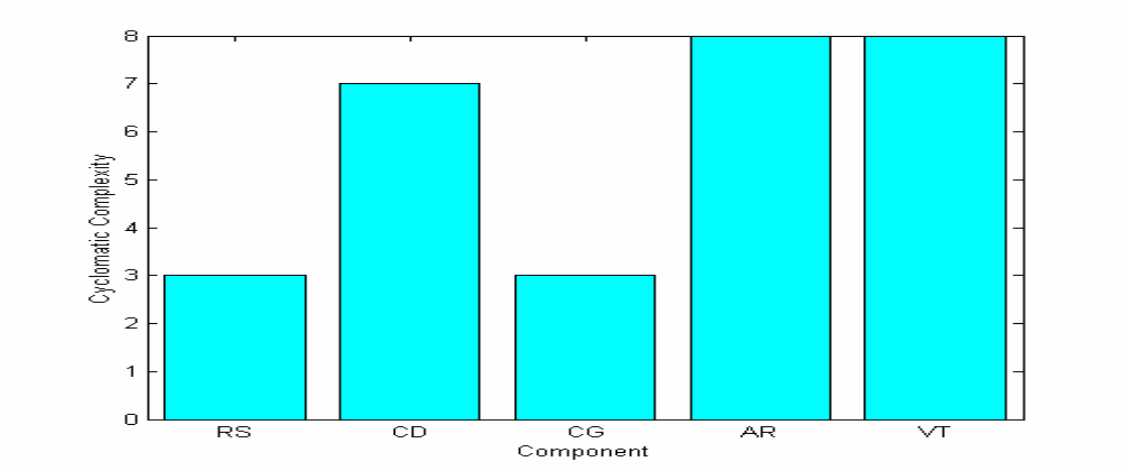


Figure 8 Pace maker cyclomatic complexity

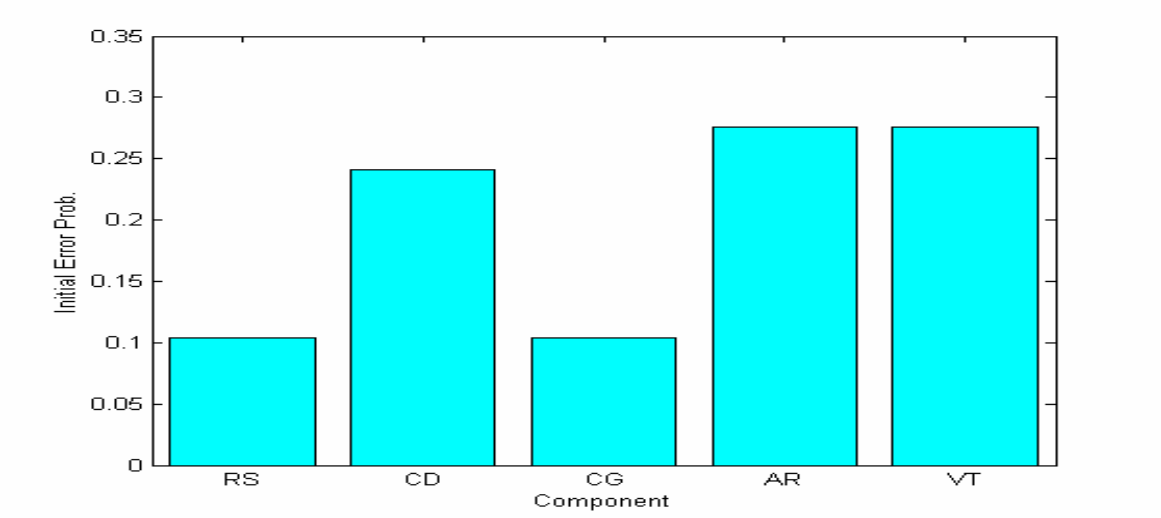


Figure 9 Pace maker initial error probability

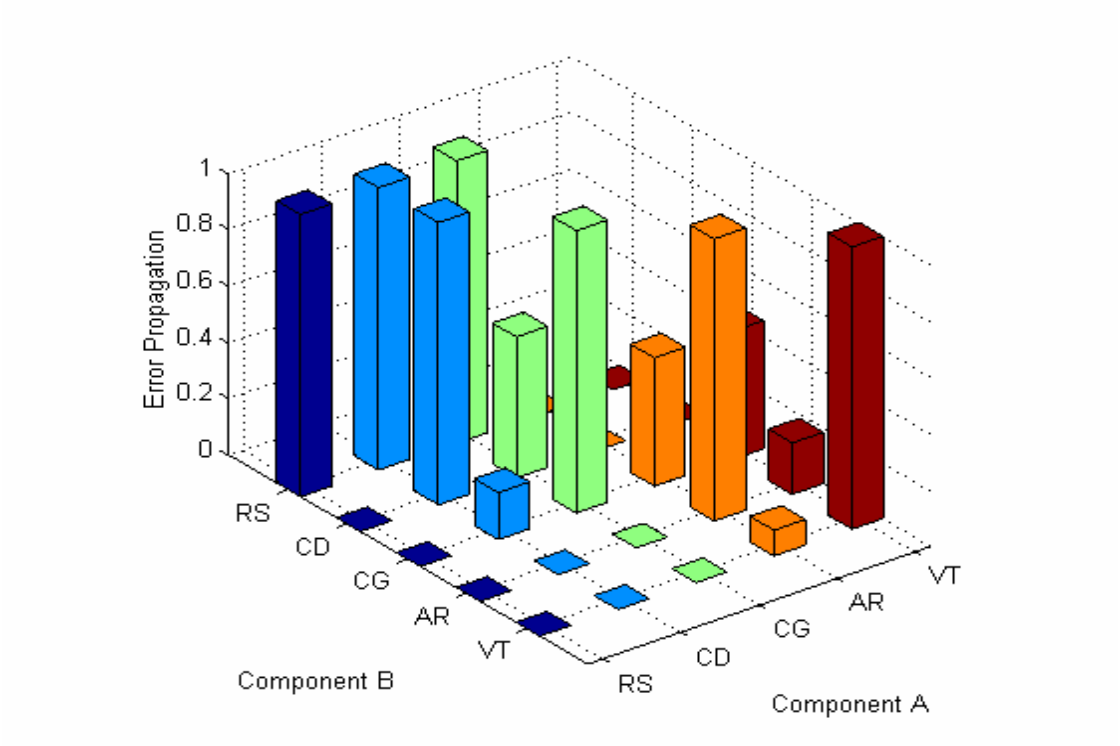


Figure 10 Pace maker error propagation matrix - analytical results

Table 9 Components severity of the pace maker case study

	Components				
	RS	**CD**	**CG**	**AR**	**VT**
Severity Level	Minor	Minor.	Major.	Catastrophic	Catastrophic

The reliability-based risk factors for the components of the pace maker system using equation (4.5) are estimated, shown in Figure 11. It also shows a comparison between components reliability-based risk factors for pace maker case study with and without considering error propagation. There is a difference between the components risk factor levels in each case. This difference becomes more significant when considering cumulative error propagation. Furthermore component CG changes its risk level ranking when considering error propagation. Due to relatively high error propagation values and high connectivity among the components of the case study, errors propagate among the components and causes failures to occur causing the components reliability-based risk factor to increase

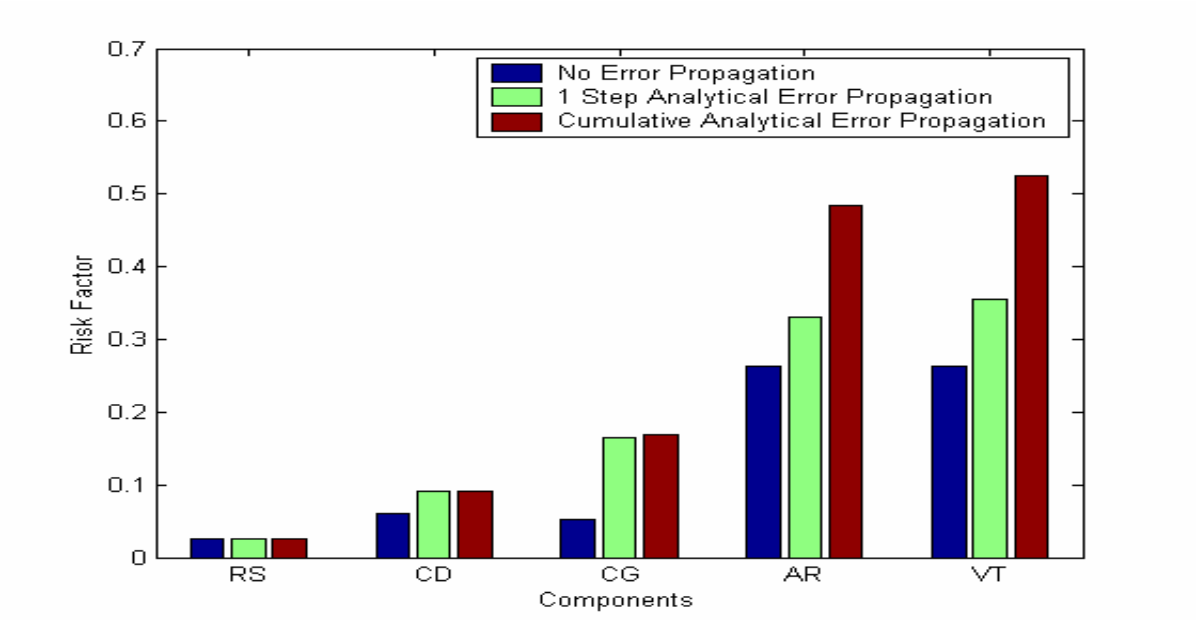


Figure 11 Comparing components reliability-based risk factors for pace maker case study

4.5.2 CM1 Case Study Results

CM1 is a software component of a data processing unit used in an instrument which exploits data to probe the early universe. This case study is from the Metrics Data Program [NASA MDP]. A UML-RT model for CM1 is constructed from the artifacts provided. (The detail of this case study is introduced in Appendix I.C). The cyclomatic complexity of the CM1 components are shown in Figure 12. By normalizing the cyclomatic complexity by their total sum, we get as an estimate for the initial error probabilities IEP, given in Figure 13.

Using the CM1 case study artifacts, the error propagation probabilities results are estimated, shown in Figure 14. The components severity levels of the CM1 case study are given in Table 10. In general, the device drivers have a catastrophic severity level, as they could be very difficult to debug them on-orbit. The application-level components are of critical severity levels, as they make use of the device drivers. The application level components usually are of major severity levels. Finally, there are components of minor severity levels as they are not mission-critical.

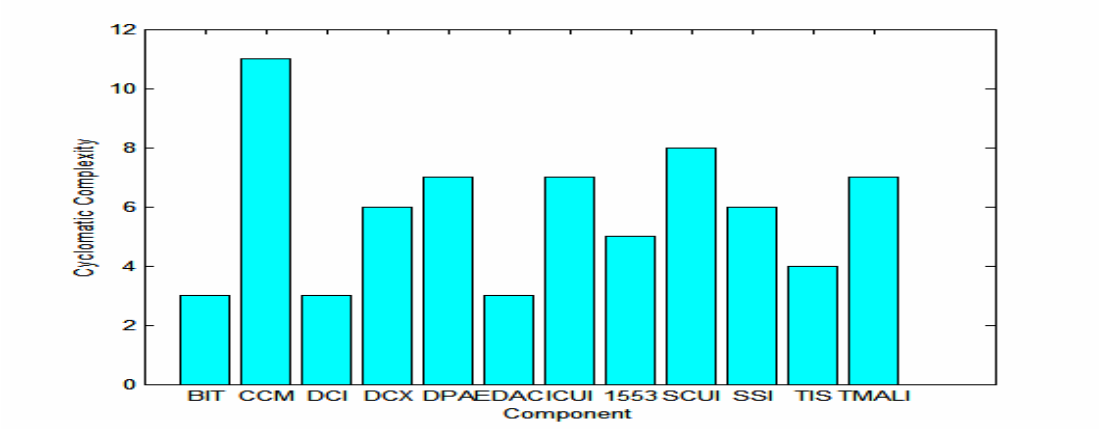


Figure 12 CM1 case study cyclomatic complexity

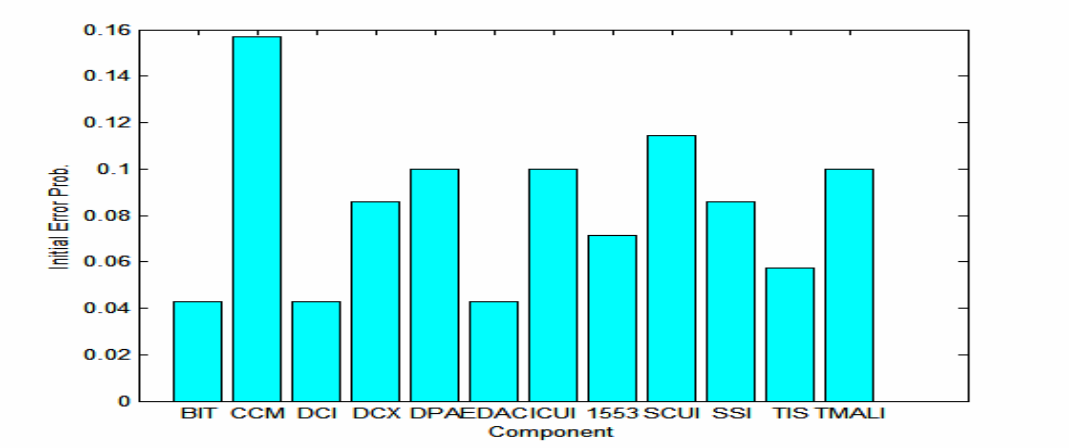


Figure 13 CM1 case study initial error probability

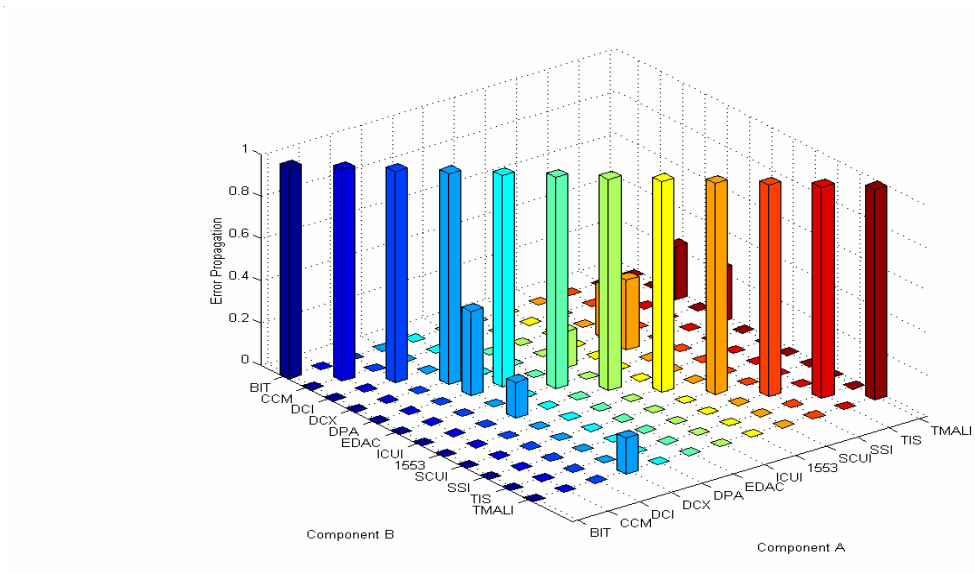


Figure 14 CM1 case study error propagation matrix - analytical results

Table 10 Components severity of the CM1 case study

	Components											
	BIT	**CCM**	**DCI**	**DCX**	**DPA**	**EDAC**	**ICUI**	**1553**	**SCUI**	**SSI**	**TIS**	**TMALI**
Severity Level	Minor	Cat.	Cat.	Minor	Major	Major	Critical	Cat.	Critical	Cat	Major	Critical

The reliability-based risk factors for the components of the CM1 system using equation (4.5) are estimated, shown in Figure 15. The Figure also shows a comparison between components reliability-based risk factors for CM1 case study when considering: no error propagation, 1-step analytical error propagation and cumulative analytical error propagation. The difference between the components risk factor levels in each case is limited to components DCX, ICUI, SCUI and TMALI. The difference is quite significant in these components and becomes more significant when considering cumulative error propagation probabilities. As result of considering error propagation in the estimation of components reliability-based risk factors, components DCX, ICUI and TMALI change their risk level ranking when considering error propagation.

It is worth noting that the estimation of the error propagation probabilities in this case study is based on a high-level model that does not capture lots of the details of the system. That results in an inaccurate estimate for error propagation of the system. As the model becomes more detailed, the error propagation estimates will be more accurate and the estimates for the risk factors for the rest of the components will differ as a result.

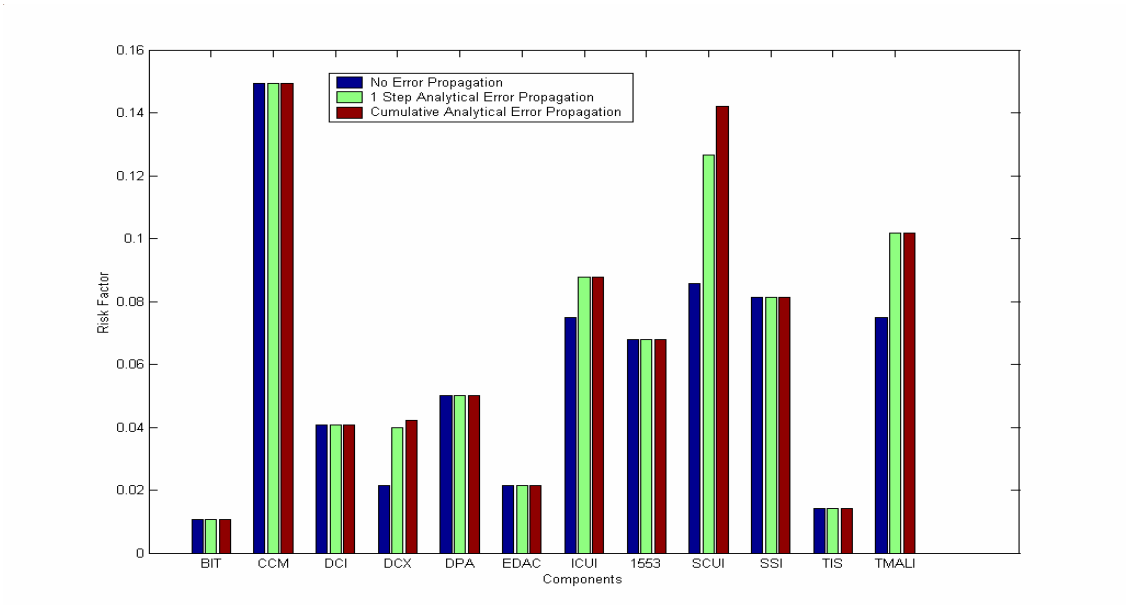


Figure 15 Comparing CM1 case study reliability risk factors

4.5.3 Command and Control System Case Study Results

To get an estimate for the initial error probabilities IEP, we normalize the cyclomatic complexity by their sum. The cyclomatic complexity of the command and control system components is shown in Figure 16. (The case study details are in appendix I.A). The initial error probabilities IEP estimates are given in Figure 17. Using the command and control case study artifacts, the error propagation probabilities results are estimated. In sections 4.1 and 4.2, we estimated the analytical one-step, empirical one-step, analytical cumulative, empirical cumulative error propagation probabilities matrices. We use these estimates to evaluate the reliability-based risk factors for the components of the system the assignment of component severity level of each component is based on the hazard analysis conducted by domain experts knowledgeable about the case study. The components severity levels of the command and control system case study are given in Table 11.

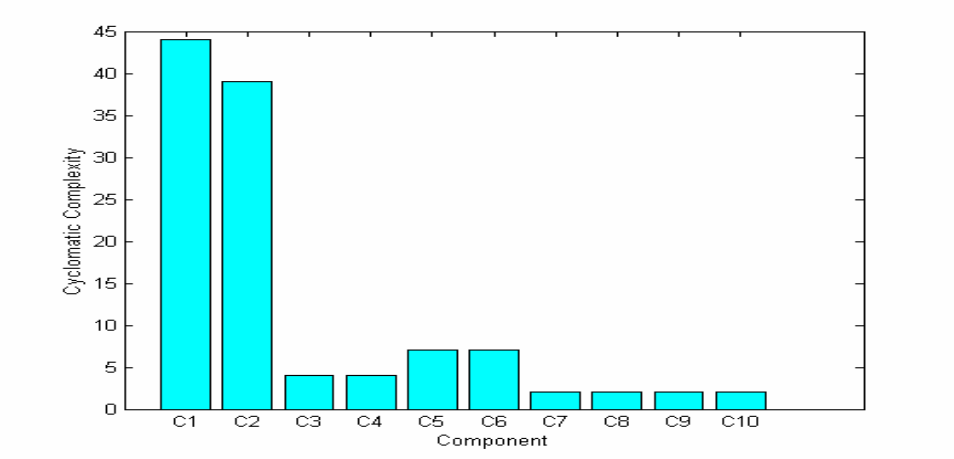


Figure 16 Command and control system cyclomatic complexity

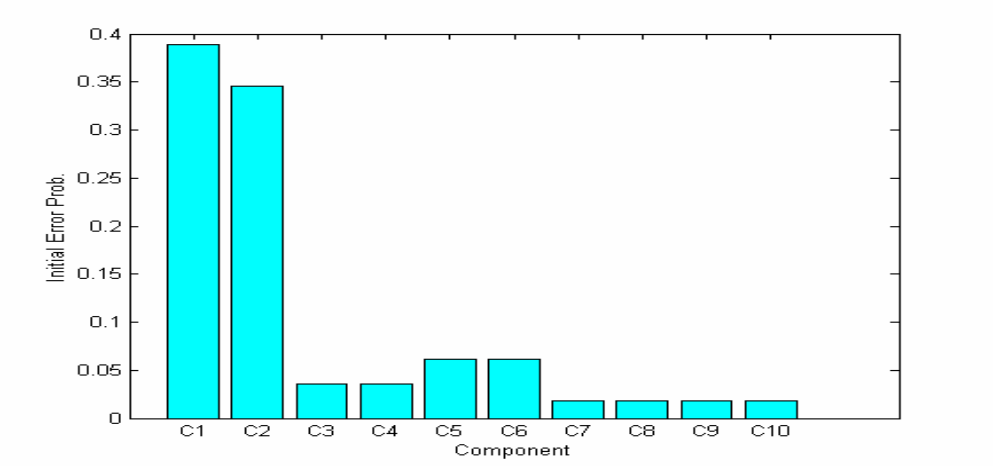


Figure 17 Command and control system initial error probability

Table 11 Components severity of the command and control case study

	Components									
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
Severity Level	Cat.	Cat.	Critical	Critical	Major	Major	Critical	Minor	Major	Minor

Figure 18 presents the reliability-based risk factors for the components of the command and control system using equation (4.5) and different error propagation estimates probabilities. The comparison shows that there is an increase in the components risk factor levels between considering error propagation and not considering it. In the case of empirical results, the increase is larger when considering cumulative error propagation than one step error propagation, as it accounts for all error propagating regardless of just being in one step. That increase is caused by relatively high error propagation values among the components of the case study. Thus, errors propagate among the components and causes failures to occur causing the components reliability-based risk factor to increase.

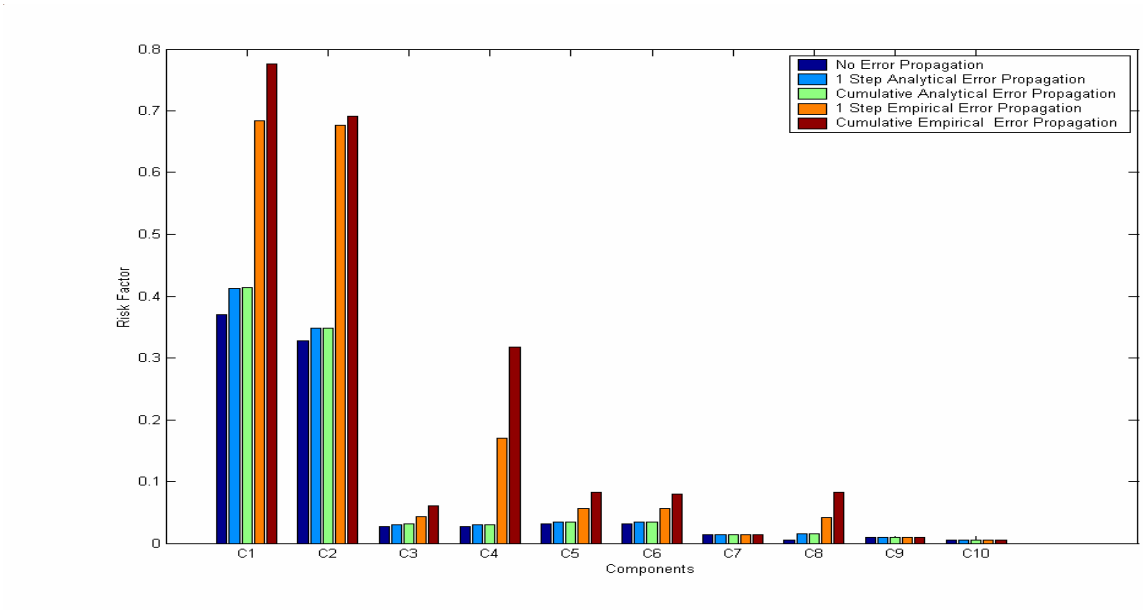


Figure 18 Comparing command and control system reliability risk factors

This case study represents the case when there is one component in the system (component C1) that manages most of the behavior of the system and controls the rest of the components in the system. Also, in the case study there is failure recovery component (component C2) that takes control in case of a failure. The rest of the components are playing just supporting roles and the whole system functionality is concentrate in these two components. This explains the huge difference in risk factor values between these

two components C1 and C2 and the rest of the components of the system. Furthermore, this architecuture causes the error propagation matrix to be sparse. Thus, errors are not capable of making further propagations in the system. Therefore, there are no significant changes in the ranking of the components of the system when considering their reliability-based risk factor, except component C4 when considering empirical results.

4.6 Summary and Discussion

In this chapter, we analytically estimated the error propagation probabilities for an industrial case study. In order to validate the analytical results, we conducted a fault injection experiment to get empirical estimates of the error propagation probabilities. We found a correlation of at least 0.55 between the two results in case of single-step error propagation probabilities and a correlation of 0.46 in the case of cumulative error propagation probabilities. Furthermore to ensure that we didn't get these correlations by chance, we computed the t statistic to examine the statistical significance for these correlations. We obtained p-values < 0.05 which shows that the correlation between experimental and analytical error propagation matrices is statistically significant.

Based on the validated results of the error propagation study, we modified the estimation methodology of components reliability-based risk to consider error propagation among the system components. The topology of the components of the system captured in terms of error propagation probabilities of these components alters their reliability-based risk level. The results obtained from the case studies show that it is important to take into consideration the specifics of the architecture of the system under investigation. As each system characteristics are quite different and that results into different reliability-based risk associated with the system. These finding agrees with [Pelànek 2004]. Pelànek explored the state space of a number of systems with model checking algorithms. The author gathered a large collection of state spaces and performed an extensive study of their structural properties. He concluded that although state spaces share some typical properties in common, some can differ significantly.

In order to get a good estimate of the error propagation probabilities, the estimation procedure requires detailed software architecture in term of detailed state diagram for each component in the system. Early in the software life-cycle, such data are usually not available. The error propagation estimation procedure can be applied if the software developers agree on investing in detailed architecture of the system and keep it up to date as the software progress in development stages. On the other hand when comparing the effort required to get the error propagation probabilities using a detailed architecture to the effort required to conduct fault injection analysis, estimating error propagation from the architecture requires much less effort.

5 Reliability-Based Risk Assessment with Functional Dependencies

In this chapter, we concentrate on reliability-based risk assessment with functional dependencies. In the context of object-oriented modeling, we deal with risk assessment estimation taking into account use-case relationships. First, we start by discussing the reliability-based risk assessment methodology. Afterward, we investigate the use-case based analysis for software system. Then, we define the use cases terminology, describe the proposed risk analysis process and present the approach for dealing with relationships between uses cases. Finally, we propose an algorithm for estimating the system risk factor from the use case model that includes relationships among use cases.

5.1 Reliability-Based Risk Assessment Methodology

For sake of completeness, this section describes the analytical modeling approach used to estimate scenario's risk factor based on the results presented in [Goseva-Popstojanova+2003]. The estimation of the heuristic risk factors of architecture elements in [Goseva-Popstojanova+2003] is based on the previous work presented in [Yacoub+ 2002] which used dynamic complexity and dynamic coupling metrics to define complexity factors for the architecture elements (components and connectors) and then combined these complexity factors with severity levels estimated using Failure Mode and Effect Analysis. In [Goseva-Popstojanova+2003], we generalized the state-based modeling approach previously used for architecture-based software reliability estimation [Goseva-Popstojanova+2001].

We start by describing the risk analysis process. Then, we describe the techniques for determining the risk factors of components and connectors in a given scenario and present a Markov model for determining scenario risk factor. Next, we present the methods used to estimate use cases and overall system risk factors.

5.1.1 The Risk Analysis Process

The use cases and scenarios of a UML specification drive the risk analysis process that we propose in this section. The proposed risk analysis process consists of the steps shown in Figure 19. We assume that the UML logical architectural model consists of a use case diagram defining several independent use cases, and that each use case is realized with one or more independent scenarios modeled using sequence diagrams.

The proposed risk analysis process iterates on the use cases and the scenarios that realize each use case and determines the component/connector risk factors for each scenario, as well as the scenarios and

use cases risk factors. For each scenario, the component (connector) risk factors are estimated as a product of the dynamic complexity (coupling) of the component (connector) behavioral specification measured from the UML sequence diagrams and the severity level assigned by the domain expert using hazard analysis and Failure Mode and Effect Analysis. Then, a Markov model is constructed for each scenario based on the sequence diagram and a scenario risk factor is determined. Further, the use cases and overall system risk factors are estimated. The outcome of the above process is a list of critical scenarios in each use case, a list of critical use cases, and a list of critical components/connectors for each scenario and each use case.

For each use case

For each scenario

For each component

Measure dynamic complexity

Assign severity based on FMEA and hazard analysis

Calculate component's risk factor

For each connector

Measure dynamic coupling

Assign severity based on FEMA and hazard analysis

Calculate connector's risk factor

Generate critical component/connector list

Construct Markov model & Calculate transition probabilities

Calculate scenario's risk factor

Rank the scenarios based on risk factors, Determine critical scenarios list

Calculate use case risk factor

Rank use cases based on risk factors, Determine critical use case list

Determine critical component/connector list in the system scope

Calculate overall system risk factor

Figure 19 The risk analysis process

5.1.2 Assessment of the Component/Connector Risk Factors

For each scenario S_x, we calculate heuristic risk factors for each component and connector participating in the scenario based on the dynamic complexity, dynamic coupling and severity level. Note that in general these values will be different for different scenarios.

The risk factor rf_i^x of a component *i* in scenario S_x is defined as

$$rf_i^x = DOC_i^x \cdot svt_i^x \tag{5.1}$$

where DOC_i^x $(0 \le DOC_i^x \le 1)$ is the normalized complexity of the i^{th} component in the scenario S_x, and svt_i^x $(0 \le svt_i^x < 1)$ is the severity level for the i^{th} component in the scenario S_x.

The risk factor rf_{ij}^x for a connector between components *i* and *j* in the scenario S_x is given by

$$rf_{ij}^x = EOC_{ij}^x \cdot svt_{ij}^x \tag{5.2}$$

where EOC_{ij}^x $(0 \le EOC_{ij}^x \le 1)$ is the normalized coupling for the connector between i^{th} and j^{th} components in the scenario S_x, and svt_{ij}^x $(0 \le svt_{ij}^x < 1)$ is the severity level for the connector between the i^{th} and the j^{th} components in the scenario S_x.

Next we describe the process of estimating the normalized component complexity DOC_i^x, normalized connector coupling EOC_{ij}^x, and severity levels for the components svt_i^x and connectors svt_{ij}^x.

5.1.2.1 Dynamic Specifications Metrics using UML

To develop risk mitigation strategies and improve software quality, we should be able to estimate the fault proneness of software components and connectors in the early design phase of the software life cycle. It is well known that there is a correlation between the number of faults found in a software component and its complexity [Munson+ 1996]. In this study we compute the dynamic complexity of state charts as a dynamic metric for components [Hassan+ 2001]. Coupling between components provides important information for identifying possible sources of exporting errors, identifying tightly coupled components, and testing interactions between components. Therefore, we compute dynamic coupling between components as a dynamic metric related to the fault proneness for connectors [Hassan+ 2001].

i) Normalized dynamic complexity of a component

We use a measure of component complexity similar to McCabe's cyclomatic complexity [McCabe 1976]. However, in contrast to McCabe's cyclometric complexity which is based on the control flow graph of the source code, our metric for component's dynamic complexity is based on the UML state charts that are available during early stages of the software life cycle. The state chart of each component *i* has a number of states and transition between these states that describe the dynamic behavior of the component. For each scenario S_x a subset of all states of component *i* are visited and a subset of all transitions is traversed. Let denote with C_i^x the subset of states for a component *i* visited in the scenario S_x and with T_i^x the subset of transitions traversed in the state chart of component *i* in the scenario S_x. The subset of states C_i^x and the corresponding transitions T_i^x are mapped into a control flow graph. The number of nodes in this graph is $c_i^x = \left| C_i^x \right|$ which is the cardinality of C_i^x. Similarly, the number of edges in this graph is $t_i^x = \left| T_i^x \right|$ which is the cardinality of T_i^x. It follows that the dynamic complexity doc_i^x of component *i* in scenario S_x is defined as

$$doc_i^x = t_i^x - c_i^x + 2. \tag{5.3}$$

The normalized dynamic complexity DOC_i^x of a component *i* in scenario S_x is obtained by normalizing the dynamic complexity doc_i^x with respect to the sum of complexities for all active components in scenario S_x

$$DOC_i^x = \frac{doc_i^x}{\sum_{k \in S_x} doc_k^x}. \tag{5.4}$$

ii) Normalized dynamic coupling of a connector

We use the matrix representation for coupling where rows and columns are indexed by components and the off-diagonal matrix cells represent coupling between the two components of the corresponding row and column [Hassan+ 2001]. The row index indicates the sending component, while the column index indicates the receiving component. Dynamic coupling metrics are calculated for active connectors during

execution of a specific scenario. We compute these metrics directly from the UML sequence diagrams by applying the same set of formulas given in [Yacoub+ 1999].

Let denote with MT_{ij}^{x} the set of messages sent from component *i* to component *j* during the execution of scenario S_x and with MT^{x} the set of all messages exchanged between all components active during the execution of scenario S_x. Then, we define the export coupling EOC_{ij}^{x} from component *i* to component *j* in scenario S_x as a ratio of the number of messages sent from *i* to *j* and the total number of messages exchanged in the scenario S_x

$$EOC_{ij}^{x} = \frac{\left|MT_{ij}^{x}\right| \quad i, j \in S_x \, , i \neq j}{\left|MT^{x}\right|} \tag{5.5}$$

5.1.2.2 Severity Analysis

In addition to the estimates of the fault proneness of each component and connector based on the dynamic complexity and dynamic coupling, for the assessment of components and connectors risk factors we need to consider the severity of the consequences of potential failures. For example, a component may have low complexity, but its failure may lead to catastrophic consequences. Therefore, our methodology takes into consideration the severity associated with each component and connector based on how their failures affect the system operation. Domain experts play a major role in ranking the severity levels. Experts estimate the severity of the components and connectors based on their experience with other systems in the same field. Domain experts can rank severity in more than one way and for more than one purpose [Bowles 1998]. According to MIL_STD_1629A, severity considers the worst case consequence of a failure determined by the degree of injury, property damage, system damage, and mission loss that could ultimately occur. Based on hazard analysis [Sundararajan 1991], we identify the following severity classes:

- *Catastrophic*: A failure may cause death or total system loss.
- *Critical*: A failure may cause severe injury, major property damage, major system damage, or major loss of production.
- *Marginal*: A failure may cause minor injury, minor property damage, minor system damage, or delay or minor loss of production.

- *Minor*: A failure is not serious enough to cause injury, property damage, or system damage, but will result in unscheduled maintenance or repair.

We assign severity indices of 0.25, 0.50, 0.75, and 0.95 to minor, marginal, critical, and catastrophic severity classes respectively. The selection of values for the severity classes on a linear scale is based on the study conducted by Ammar *et.al.* [Yacoub+ 2000]. However, other values could be assigned to severity classes, such as for example using the exponential scale.

5.1.3 Scenarios Risk Factors

We use an analytical modeling approach to derive the risk factor of each scenario. For this purpose we generalize the state-based modeling approach previously used for architecture-based software reliability estimation [Goseva-Popstojanova+2001]. Thus, the software reliability model first published in [Cheung 1980] considers only component failures. In the scenario risk model we account for both component and connector failures, that is, consider both component and connector risk factors. In addition, instead of a single failure state for the scenario, we consider multiple failure states that represent failure modes with different severity. This approach allows us to derive not only the overall scenario risk factor, but also its distribution over different severity classes which provides additional insights important for risk analysis. For example, the two scenarios may have close values of scenarios risk factors with significantly different distributions among severity classes. Then, it can be inferred that the scenario with a risk factor distributed among more severe failure classes (e.g., critical and catastrophic) deserves more attention than the other scenario.

The scenario risk model is developed in two steps. First, a control flow graph that describes software execution behavior with respect to the manner in which different components interact is constructed using the UML sequence diagrams. It is assumed that a control flow graph has a single entry (*S*) and a single exit node (*T*) representing the beginning and the termination of the execution, respectively. Note that this is not a fundamental requirement. The model can easily be extended to cover multi-entry, multi-exit graphs.

The states in the control flow graph represent active components, while the arcs represent the transfer of control between components (i.e. connectors). It is further assumed that the transfer of control between components has a Markov property which means that given the knowledge of the component in control at any given time, the future behavior of the system is conditionally independent of the past behavior. This assumption allows us to model software execution behavior for scenario S_x with an absorbing discrete time Markov chain (DTMC) with a transition probability matrix $P^x = [p_{ij}^x]$, where p_{ij}^x is interpreted as

the conditional probability that the program will next execute component j, given that it has just completed the execution of the component i. The transition probability from component *i* to component *j* in scenario S_x is estimated as $p_{ij}^x = \frac{n_{ij}^x}{n_i^x}$, where n_{ij}^x is the number of times messages are transmitted from component *i* to component *j* and $n_i^x = \sum_j n_{ij}^x$ is the total number of massages from component *i* to all other components that are active in the sequence diagram of the scenario S_x.

The second step of building the scenario risk model is to consider the risk factors of the components and connectors. Failure can happen during the execution period of any component or during the control transfer between two components. It is assumed that the components and connectors fail independently. Note that this assumption can be relaxed by considering higher order Markov chain [Goseva-Popstojanova+2001].

In architecture-based software reliability models [Cheung 1980], [Goseva-Popstojanova+2001] a single state *F* is added representing the occurrence of a failure. Because the severity of failures plays an important role in the risk analysis, in this work we add *m* failure states that represent failure modes with different severity. In particular, since for the pacemaker case study we consider four severity classes for each failure we add four failure states to the DTMC: F_{minor}, F_{marginal}, F_{critical}, and $F_{\text{catastrophic}}$. The transformed Markov chain, which represents the risk model of a given scenario has $(n+1)$ transient states (*n* components and a starting state *S*) and $(m+1)$ absorbing states (*m* failure states for each severity class and a terminating state *T*).

Next, we modify the transition probability matrix P^x to $\overline{P^x}$ as follows. The original transition probability p_{ij}^x between components i and j is modified into $(1-rf_i^x)\cdot p_{ij}^x \cdot (1-rf_{ij}^x)$ which represents the probability that the component i does not fail, the control is transferred to component j, and the connector between component i and j does not fail. The failure of component i is considered by creating an arc to the failure state associated with a given severity with transition probability rf_i^x. Similarly, the failure of a connector between the components i and j is considered by creating an arc to failure state associated with a given severity with transition probability $(1-rf_i^x)\cdot p_{ij}^x \cdot r_{ij}^x$. The transition probability matrix of the transformed DTMC, $\overline{P^x}$, is then partitioned so that

$$\overline{P^x} = \begin{bmatrix} Q^x & C^x \\ 0 & I \end{bmatrix} \tag{5.6}$$

where Q^x is an $(n+1)$ by $(n+1)$ sub-stochastic matrix (with at least one row sum less than 1) describing the probabilities of transition only among transient states, I is an $(m+1)$ by $(m+1)$ identity matrix and C^x is a rectangular matrix that is $(n+1)$ by $(m+1)$ describing the probabilities of transition from transient to absorbing states. We define the matrix $A^x = [a_{ik}^x]$ so that a_{ik}^x denotes the probability that the DTMC starting with a transient state i eventually gets absorbed in an absorbing state k. Then it can be shown that [Trivedi 2002]

$$A^x = (I - Q^x)^{-1} C^x. \tag{5.7}$$

Since in our case we assume a single starting state *S*, the first row of matrix A^x gives us the probabilities that DTMC is absorbed in absorbing states *T,* F_{minor}, F_{marginal}, F_{critical}, and $F_{\text{catastrophic}}$. In particular, a_{11}^x is equal to $(1 - rf^x)$, where rf^x is the scenario risk, while a_{12}^x, a_{13}^x, a_{14}^x, and a_{15}^x give us the distribution of the scenario risk factor among minor, marginal, critical, and catastrophic severity classes respectively.

5.1.4 Use Cases and Overall System Risk Factors

The risk factor rf_k of each use case U_k is obtained by averaging the risk factors of all scenarios S_x that are defined for that use case

$$rf_k = \sum_{\forall S_x \in U_k} rf^x \cdot p_k^x \tag{5.8}$$

where rf^x is the risk factor of scenario S_x in use case U_k and p_k^x is the probability of occurrence of scenario S_x in the use case U_k. Since in the pacemaker example we considered one scenario per use case, the use case risk factors are identical to the scenarios risk factors.

Similarly, the overall system risk factor is obtained by averaging the use case risk factors

$$rf = \sum_{\forall U_k} rf_k \cdot p_k \tag{5.9}$$

where rf_k and p_k are the risk factor and probability of occurrence of the use case U_k.

It is obvious from equations (5.8) and (5.9) that the use cases and overall system risk factors depend on the probabilities of scenarios occurrence p_k^x in the use case U_k and the probability of use case occurrence p_k. Hence, scenarios (use cases) with high risk factors but very low probability of occurrence will not contribute significantly to the overall system risk factor.

5.2 The Use-Case Based Analysis

When a large complex system is modeled, a significant amount of requirements information is represented in the use case model. A use case is defined as a set of sequences of actions the system is required to perform in order to get an observable result of value to an actor [Armour+ 2001]. When a use case model is defined, commonality among the use cases is likely to be discovered and shared. Possible extensions and additional behavior may also be uncovered and defined. Some use cases may contain behaviors that are similar in many aspects. To effectively design a use case model that avoids redundant use cases, UML includes the use of relationships between use cases. Modelers who develop a use case model without showing relationship between the use cases are not modeling the system properly. In real software systems, use cases model different requirements that the system should provide and these requirements are typically dependent on each other. These dependencies must be modeled as use case relationships.

Before we proceed with our approach, we present the UML use case terminology and the basic definitions. The *base use case* captures the behavior and interactions that occur between the actors and the system within the use case flow of events [OMG 2005]. It provides an excellent viewpoint on the overall system behavior. The addition of alternative flow descriptions and conditional logic helps to define the variation and exceptions within a use case. Use case modeling provides a number of constructs that support the clear elaboration of added complexity and details of the relationships between use cases [Armour+ 2001], such as:

- **<<Extend>>** relationship models significant extensions and behavior that can occur as additions to the base use case. Thus, as shown in Figure 20 the flow of events of the base use case can be

extended by the flow of events in the extended use case. The actual control transfer in this kind of extension is optional, i.e., it takes place when a conditional guard is satisfied. Then, the flow of events returns to the same extension point in the flow of events in the base use case.

- **<<Include>>** relationship models encapsulated behavior that can be inserted into a use case and possibly reused across multiple use cases. As shown in Figure 21, the included behavior is always exercised, that is, no conditional guard is checked.

The examples shown in Figure 20 (b) and Figure 21 (b) illustrate the difference between the <<extend>> and <<include>> relationships. Thus, planning an itinerary might or might not include purchasing a ticket. On the other side, each attempt to register courses includes user validation.

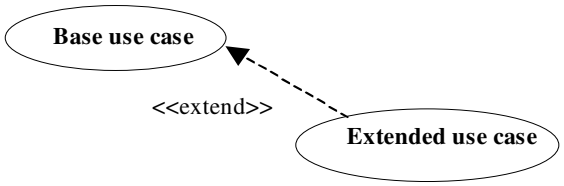
5.3 Risk Assessment Methodology with Functional Dependencies

In this section, we start by defining the terminology. Then, we describe the proposed risk analysis process and present the approach for dealing with relationships between uses cases. Finally, we propose an algorithm for estimating the system risk factor from the use case model that includes relationships among use cases.

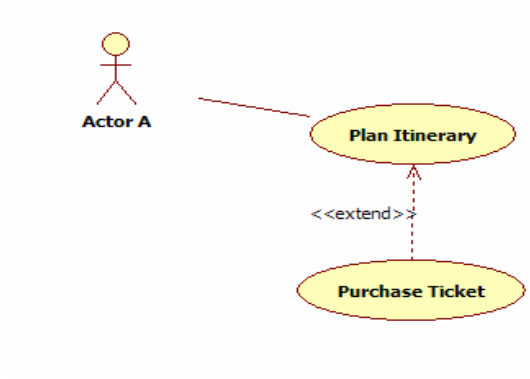
5.3.1 Use Cases Terminology Used

To come up with a systemic approach for dealing with relationships between use cases, we introduce the following terminology.

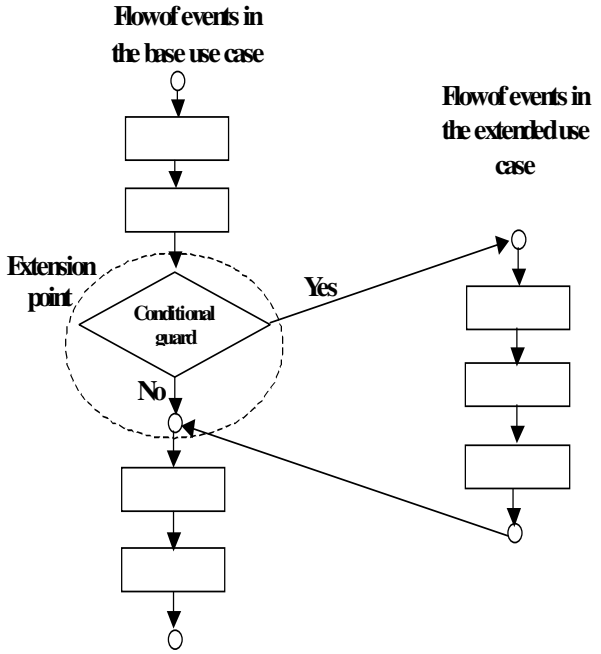
- *Primitive Use Case (PUC)* is a use case that is not extended and does not include any other use cases. The risk factor of a primitive use case is calculated directly from its sequence diagrams as outlined in the next section. In Figure 20 (b) and Figure 21 (b), the use cases Purchase Ticket and Validate User are examples of primitive use cases.
- *Non-Primitive Use Case (NPUC)* is a base use case related to other primitive or non-primitive use case(s) by <<extend>> and/or <<include>> relationship(s). The realization of a *NPUC* must have at least one dependent scenario that depends on a scenario of a related use case. Figure 23 shows a sequence diagram of a dependent scenario where the related scenario `Purchase Ticket` mentioned at the bottom of Figure 23 is shown in Figure 24, the risk factor of a non-primitive use case cannot be calculated directly. In the following sections, we describe the way to estimate the risk factor of a non-primitive use case and present a general algorithm which is used to calculate the system risk factor based on the use case diagram that includes relationships among use cases.



a) UML notation

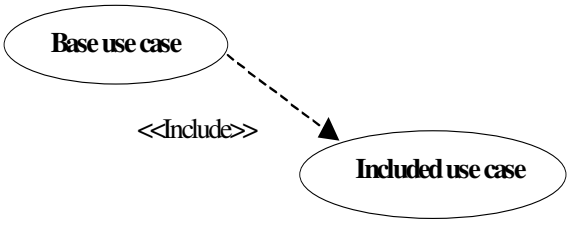


b) Example

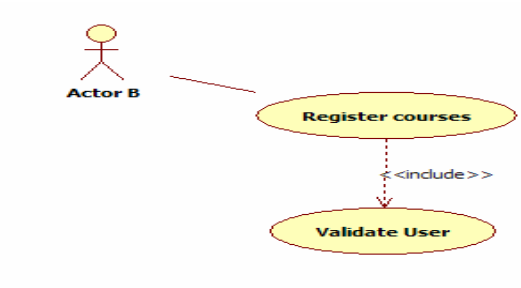


- c) Flow of events

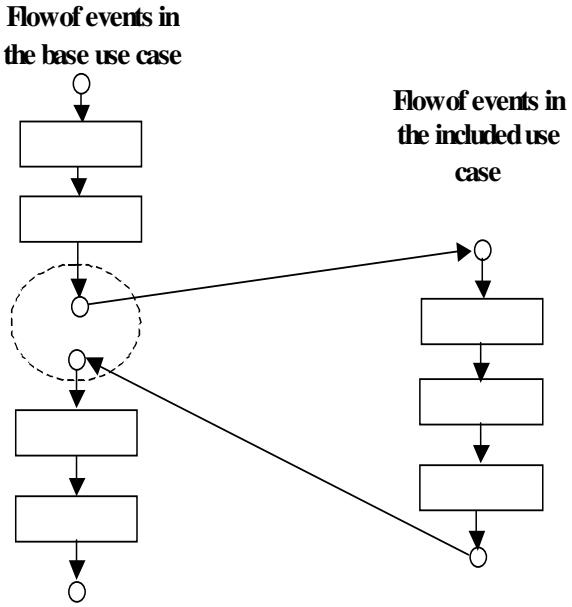
Figure 20 <<Extend>> relationship



a) UML notation



b) Example



c) Flow of events

Figure 21 <<Include>> relationship

- *Terminal Use Case (TUC)* is a use case directly associated with an actor. Terminal use cases can be either primitive or non-primitive.

In Figure 20 (b) and Figure 21 (b), the use cases Plan Itinerary and Register Courses are both non-primitive and terminal use cases since they are related to other use cases and they are also directly associated with actors.

5.3.2 Estimating the Risk Factor of Use Cases

In this section, we discuss how to estimate the risk factors of primitive and non-primitive use cases from UML models.

5.3.2.1 Primitive Use Cases

We use the estimation of scenario's risk factor to estimate the risk factor of primitive use cases. The scenario risk model is developed in two steps. First, a discrete time Markov chain (DTMC) which describes software execution behavior with respect to the manner in which different components interact is constructed using the UML sequence diagram of a use case. The transition probability matrix of this DTMC is calculated based on the frequency of message exchange. The right part of Figure 25 shows an example of a DTMC of the UML sequence diagram shown in Figure 24.

The second step of building the scenario risk model is to consider the risk factors of the components and connectors. Failure can happen during the execution period of any component or during the control transfer between two components. It is assumed that the components and connectors fail independently. The DTMC of the software execution behavior of the scenario is transformed by adding the failure severity states which represent failure modes with Minor, Marginal, Critical, and Catastrophic severity. Figure 27 and Figure 28 show a sequence diagram of the case study described in section A and the corresponding DTMC with the failure severity states. The DTMC is then solved to estimate the risk factor of the scenario and its distribution among Minor, Marginal, Critical and Catastrophic severity classes as shown in Section 5.1.1.

5.3.2.2 Non-Primitive Use Cases

In this section we propose a method that can be used to estimate the risk factor of a non-primitive use case which is related to a primitive use case by either <<extend>> or <<include>> relationship. Then, in the next section, we propose an algorithm that allows us to estimate the system risk factor from a general use case diagram that may include many different <<extend>> and <<include>> relationships among use cases.

The method for estimating the risk factor of a non-primitive use case related to primitive use cases consists of aggregating the risk factor of each primitive use case into the calculation of the risk factor of the non-primitive use case [Abdelmoez+ 2003]. For this purpose, we first estimate the risk factor of the primitive use cases. That is, we develop a discrete time Markov chains (DTMCs) that represent the primitive use cases and solve to estimate the risk factors. Each of these factors are then aggregated in the DTMC of the non-primitive use case as a single state with the corresponding estimated risk factor, as shown in Figure 22. The value of the transition probability which leads to this aggregated state that represents a primitive use case depends on the type of relationship. Thus, in the case of <<extend>> relationship, we assign probability $0<p<1$ to the transition probability leading to the state that represents the primitive use case. For <<include>> relationships, the probability of making transition to the primitive use case is set to one $(p=1)$ because it is always part of the non-primitive use case.

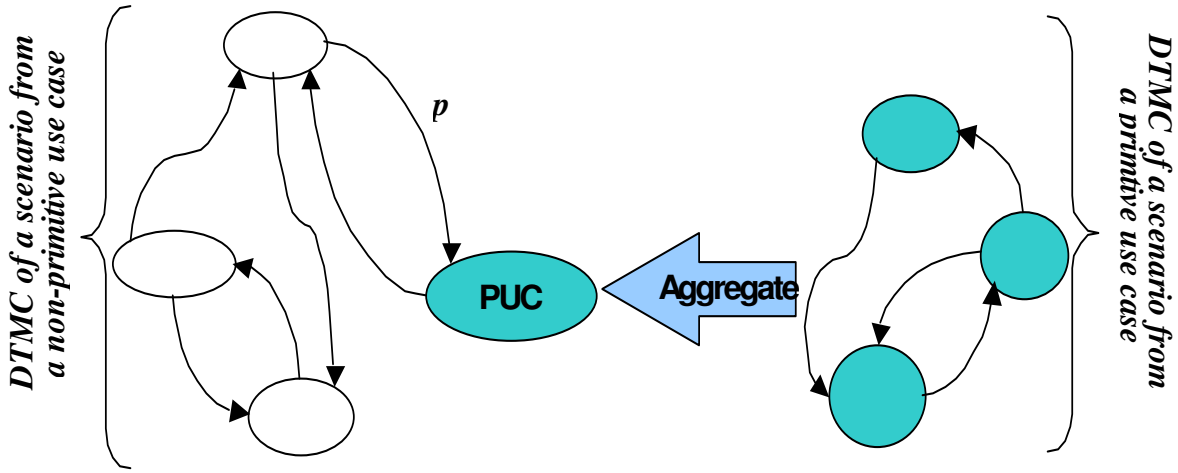


Figure 22 Dealing with use case relationships

To illustrate this process, let us consider the simple sequence diagrams shown in Figure 23 and Figure 24 which describe the interactions in the `Plan Itinerary` and `Purchase Ticket` use cases from the simple use case diagram shown in Figure 20 (b). The DTMC of the `Plan Itinerary` and `Purchase Ticket` use cases developed from theses sequence diagrams are shown in Figure 25.

After the risk factor of the primitive `Purchase Ticket` use case is estimated, it is aggregated in the `Purchase Ticket` state of the non-primitive `Plan Itinerary` use case. Because the `Plan Itinerary` use case is extended by the `Purchase Ticket` use case, the probability p of execution of the `Purchase Ticket` use case is assigned to the arc leading to the state representing it in the DTMC of the `Plan Itinerary` use case.

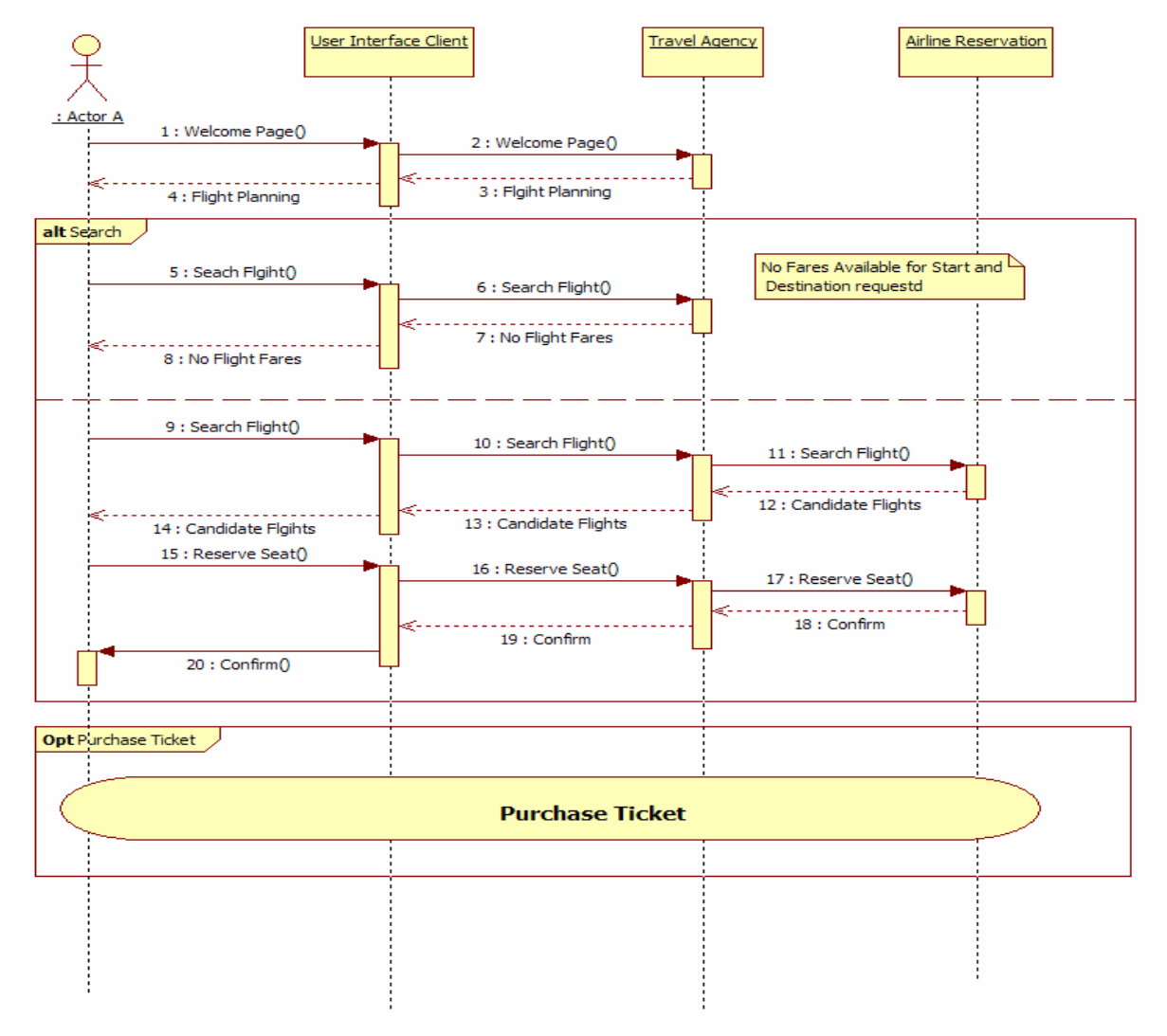


Figure 23 `Plan Itinerary` sequence diagram

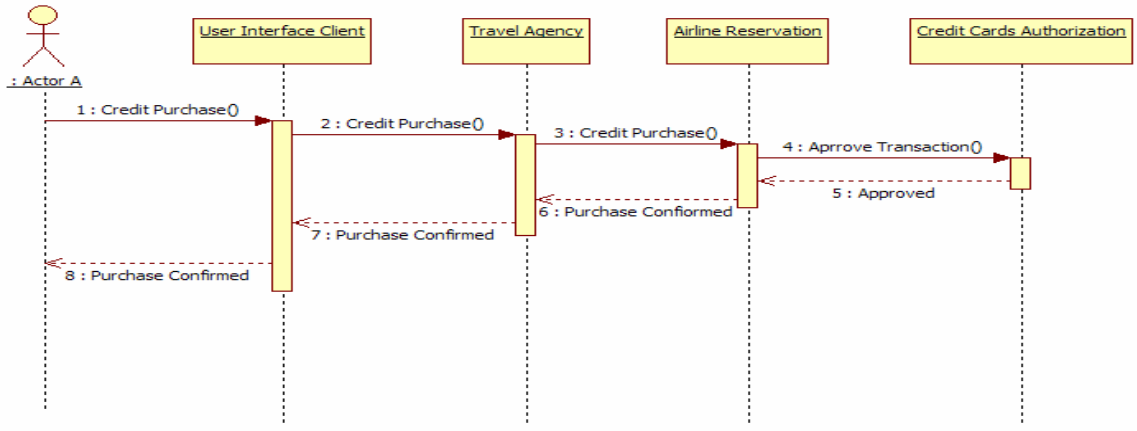


Figure 24 `Purchase Ticket` sequence diagram

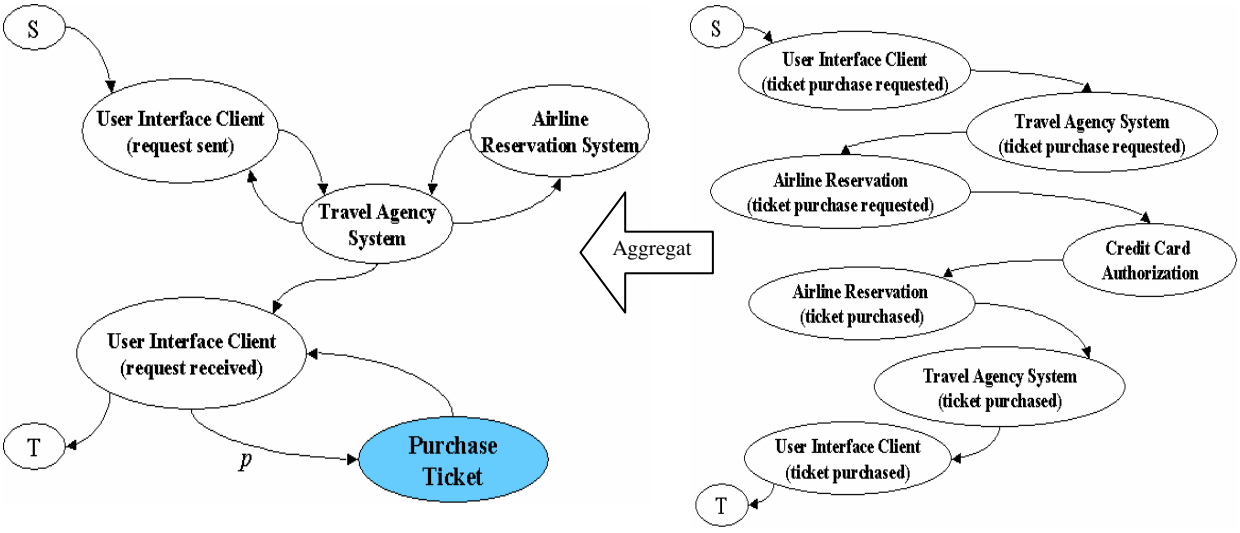


Figure 25 DTMCs of the `Plan Itinerary` and `Purchase Ticket` use cases

In general, use case diagrams are much more complex than the example given in Figure 20 (b). For example, the `Purchase Ticket` use case may be extended with `Rent a Car` use case, in which case `Plan Itinerary` and `Purchase Ticket` both are non-primitive use cases, while `Rent a Car` use case is a primitive use case. In the next section we propose an algorithm which uses recursively the method for estimation of the risk factor of a non-primitive use case related to a primitive use case by either <<extend>> or <<include>> relationship presented in this section and allows us to estimate the use cases and system risk factor of a complex use case diagram.

5.4 Algorithm for System Risk Estimation

The algorithm proposed uses an annotated use case diagram as input. We treat the use case diagram as a graph, where nodes represent use cases and arcs represent relationships between use cases. Arcs are annotated with probability $0<p<1$ in case of <<extend>> relationship or with probability $p=1$ in case of <<include>> relationship. The algorithm traverses the use case diagram, identifies use cases according to relationships between them, and then aggregates the risk factors of the use cases according to the relationships. An outline of the risk estimation algorithm is shown in Figure 26.

Our algorithm works in two passes. In the first pass the algorithm traverses the use case diagram using a depth first priority. The main outcomes of the first pass are to identify the use cases based on the relationships and to color them accordingly and to estimate the risk factors of the primitive use cases. In the second pass the algorithm traverses the colored use case diagram starting from the actors and estimates the risk factor of each terminal use case (i.e., use case directly connected to an actor) recursively Finally,

the system risk factor is calculated as a sum of risk factors of terminal use cases multiplied by the corresponding execution probabilities. An implementation for this risk aggregation algorithm considering one scenario per use case was developed by Ajith and Venu. The implementation details are given in [Guedem 2004].

```
Determine terminal, primitive, and non-primitive use cases and
their dependent scenarios

For each primitive use case
        Determine the risk factor of each scenario
        Determine the use case risk factor by taking the max risk
        scenario

For each terminal use case
        For each dependent scenario
                Recursively determine the risk factor of the
                scenario
        Determine the use case risk factor by taking the max risk
        scenario

Determine the system risk using a weighted sum of the risk
factors of the terminal use cases
```

Figure 26 Outline of the risk estimation algorithm

5.5 Command and Control System Case Study Results

One of the contributions is the application of our risk assessment methodology on an industrial case study of a command and control system used in a mission-critical application. (The details of the case study are presented in Appendix I).A. In the following subsections, the case study results are presented.

5.5.1 Scenario Risk Factors

The process of building and solving the scenario risk model is illustrated on the `Retry_Both_Pumps` scenario. As mentioned in section 5.3.2.1, the first step is to develop a control flow graph that describes software execution behavior with respect to the manner in which different

components interact using the UML sequence diagram of a scenario. Thus, the white states of the DTMC shown in Figure 28 represent the software execution behavior which corresponds to the sequence diagram of the `Retry_Both_Pumps` scenario shown in Figure 27. The transition probability matrix of this DTMC is calculated based on the frequency of message exchange as described in [Goseva-Popstojanova+2003].

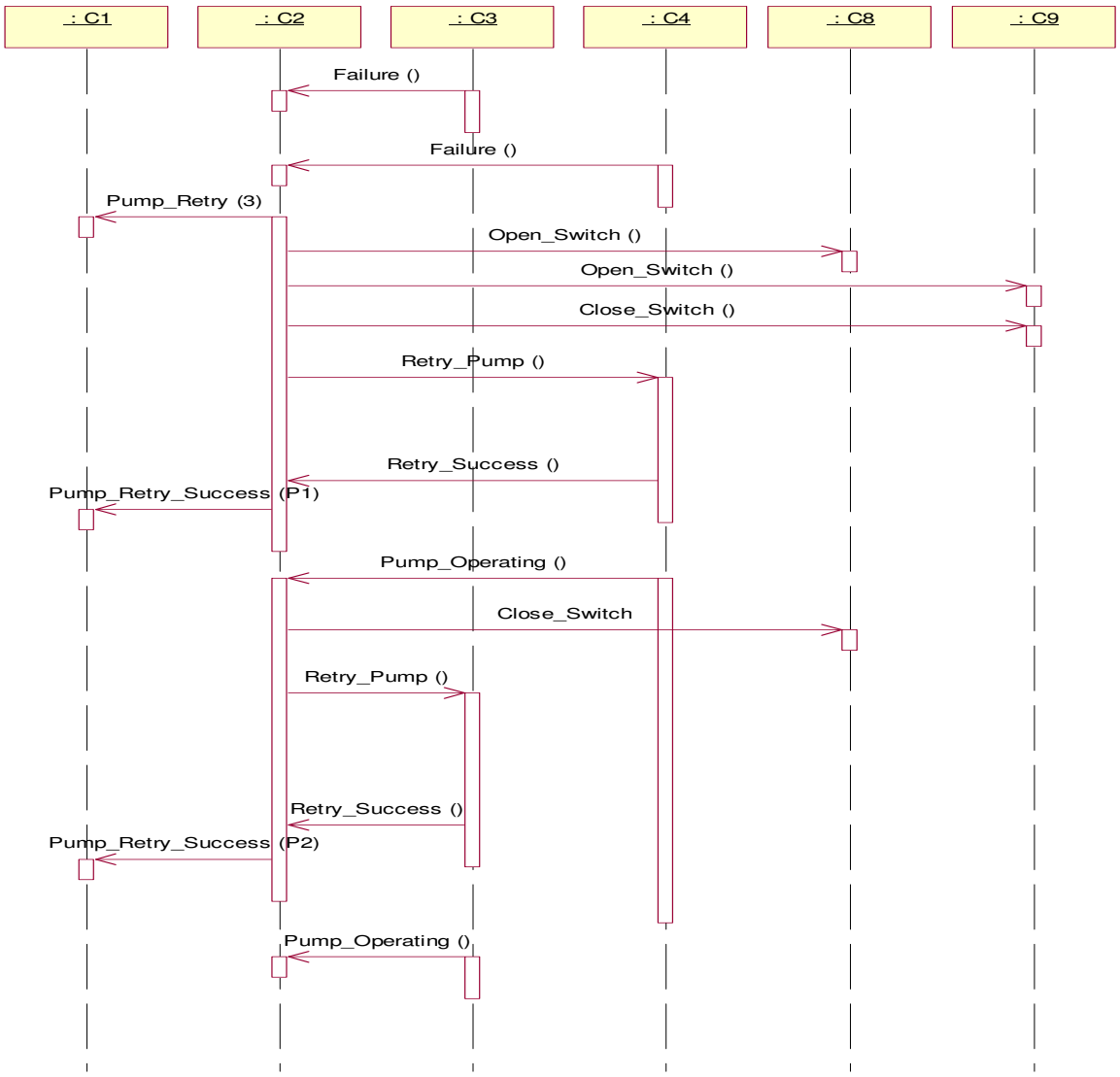


Figure 27 Sequence diagram of the `Retry Both Pumps` scenario

The second step of building the scenario risk model is to consider the risk factors of the components and connectors. The DTMC of the software execution behavior (white states only) of the `Retry_Both_Pumps` scenario is transformed by adding the dark states which represent failure modes

with Minor, Marginal, Critical, and Catastrophic severity (see Figure 28). The estimated risk factor of the `Retry_Both_Pumps` scenario is *0.7605*. This risk factor is distributed among Minor, Marginal, Critical and Catastrophic severity classes (*0.0000, 0.1100, 0.0703* and *0.5802* respectively).

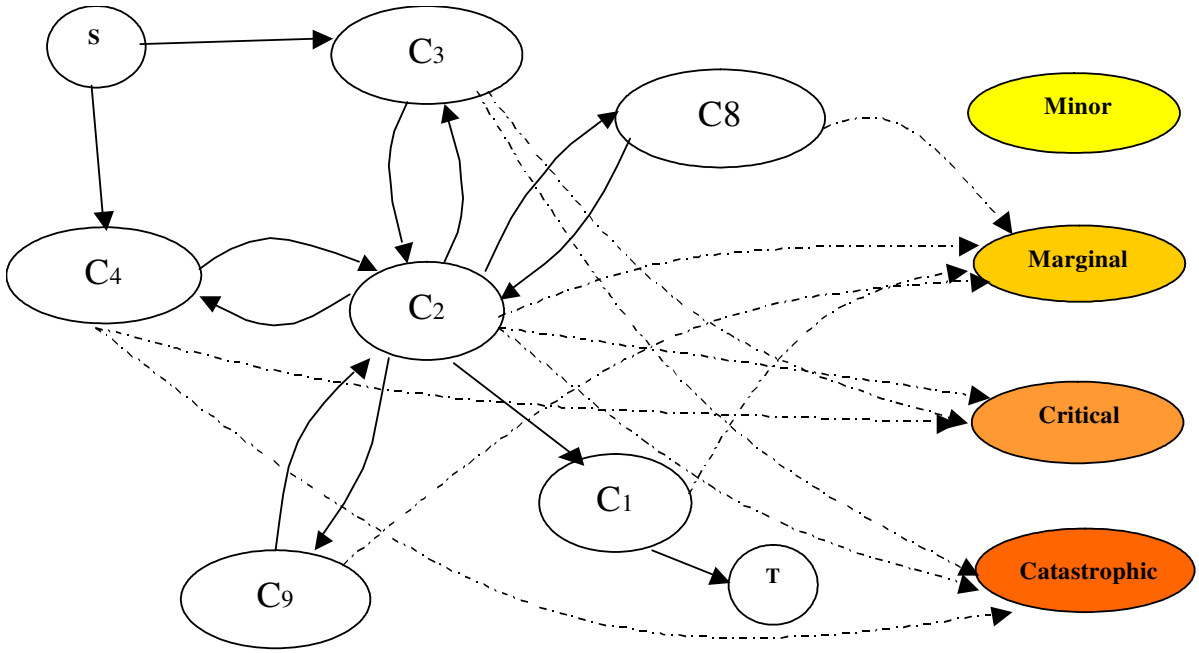


Figure 28 Risk model of the `Retry_Both_Pumps` scenario

5.5.2 Use Case and System Level Risk Factors

Next, we illustrate how the use case and system level risk factors are estimated using the generalized risk assessment methodology. As shown in Figure 96, the use case diagram of the Internal Thermal Control subsystem has many <<extend>> relationships between use cases. As presented in Section 5.3, we first estimate the risk factors of mode setting use cases and pump retry use cases, as they are the primitive ones. The risk factors of theses primitive use cases are shown in Figure 29. Then, we aggregate the risk factors of these primitive use cases to estimate the risk factors of the non-primitive use cases. Using the scenarios of Failure_Recovery and Mode_Setting non-primitive use cases, the DTMC of these use cases are constructed. Then, we embed the risk factors of the primitive use cases in the DTMC of the non-primitive use case as described in Section 5.3.2. Since the pattern of extending these non-primitive use cases is not known, we assume the probability p to be equal. Thus, the risk factors of the non-primitive use cases `Failure_Recovery` and `Mode_Setting` over different severity classes (i.e., Minor, Marginal, Critical, and Catastrophic) are evaluated.

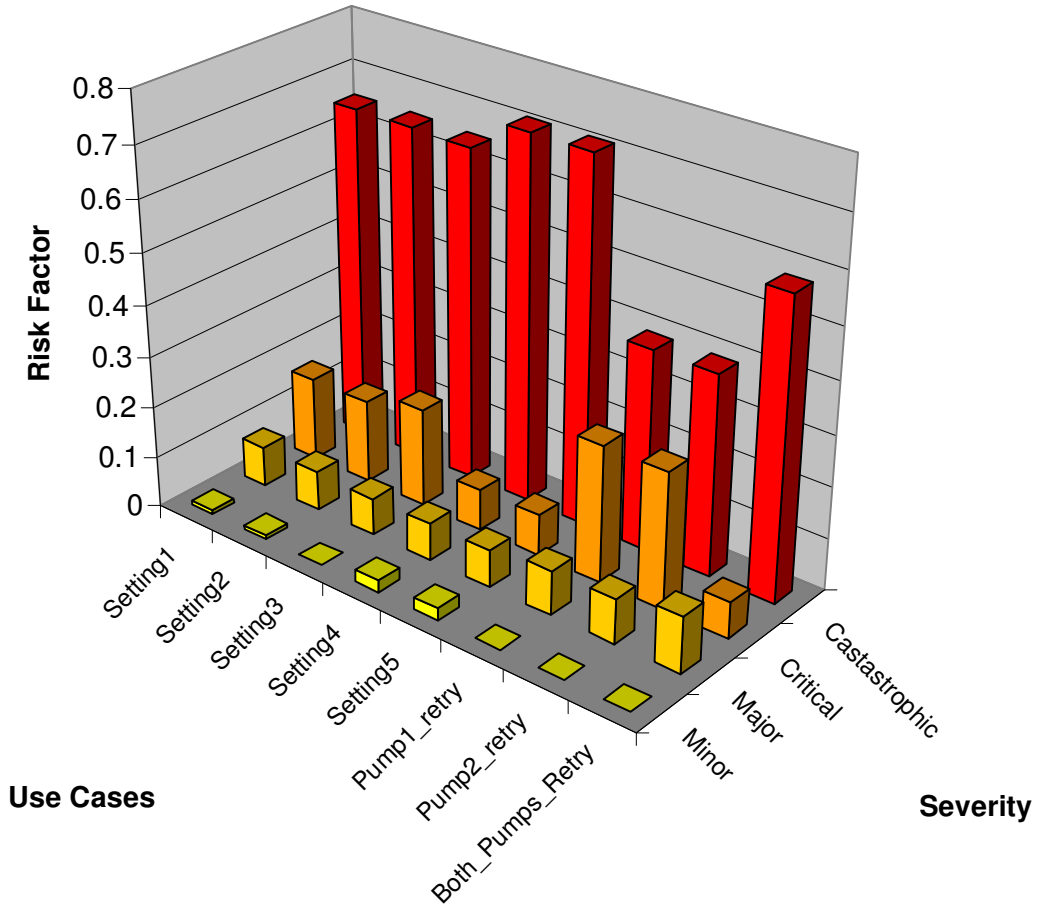


Figure 29 Risk factors of the primitive use cases

From the domain knowledge , we know that the `Failure_Recovery` use case is executed (i.e., extends the `Monitoring` use case) with probability 0.01. To estimate the risk factor of the non-primitive use case `Monitoring`, we build the DTMC of the scenario in the `Monitoring` use case and embed with probability p=0.01 the non-primitive use case `Failure_Recovery`, as shown in Figure 30. Sovling the DTMC, we get the risk factor of `Monitoring` use case.

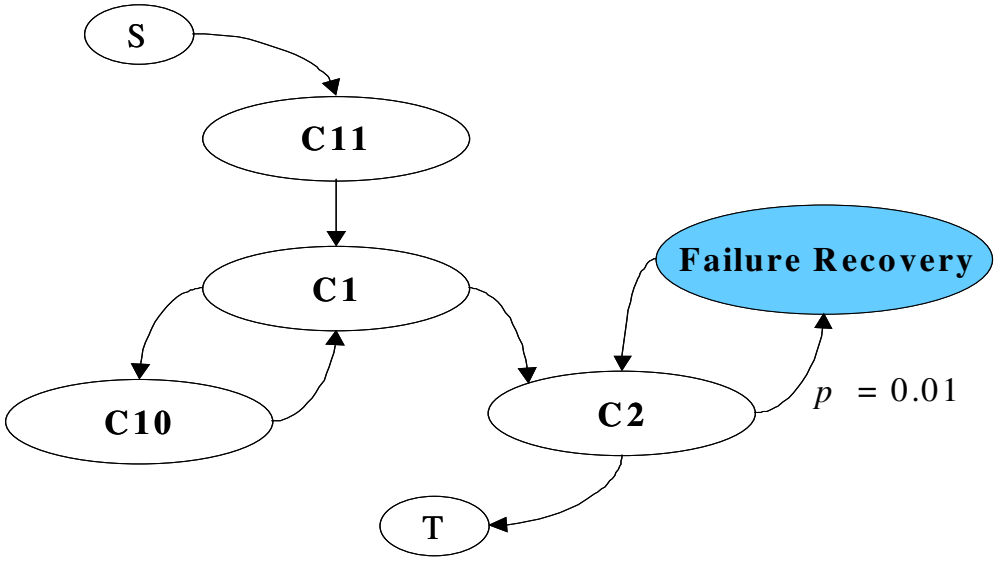


Figure 30 DTMC of the `Monitoring` use case

In Figure 31, the risk factors of the terminal use cases `Monitoring` and `Mode_Setting` are presented. The bars represent the total risk factors of these use cases along with the distribution of the risk factors over the severity classes. The `Mode_Setting` use case is riskier than the `Monitoring` use case. Even more, it has higher risk in the catastrophic severity class. Therefore, more attention should be given to the development and testing of `Mode_Setting` use case, as it is more critical than the `Monitoring` use case.

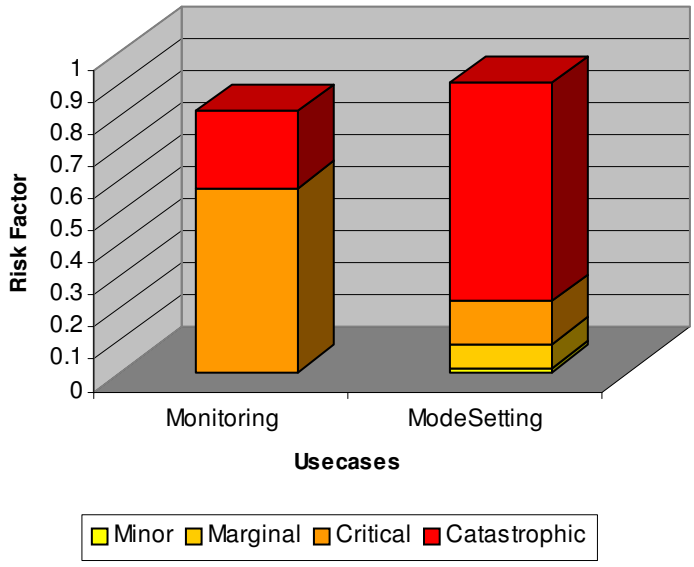


Figure 31 The risk factor of `Monitoring` and `Mode_Setting` use cases.

The overall risk factor of the Internal Thermal Control subsystem is obtained by multiplying the risk factors of the terminal use cases (`Monitoring and Mode_Setting`) with the corresponding probabilities of execution of these use cases (0.95 and 0.05, respectively). Its value is *0.8189* and it is distributed over the severity classes as shown in Figure 32.

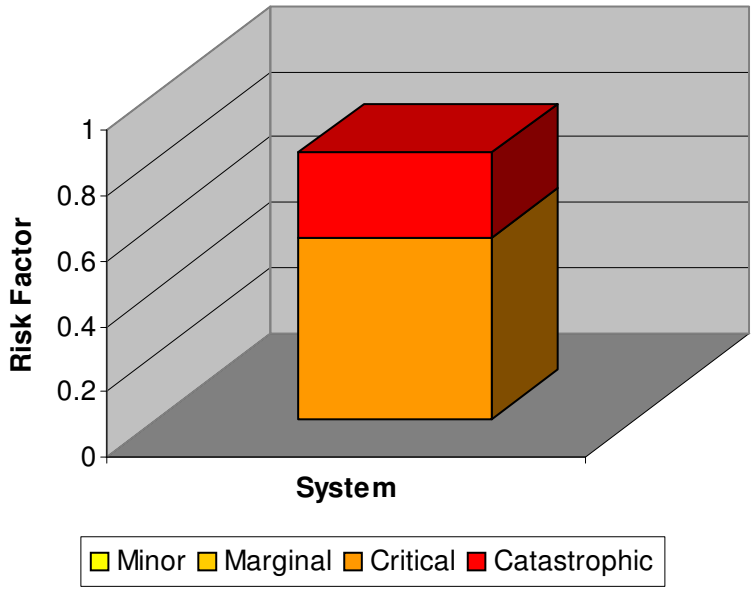


Figure 32 Distribution of the overall system risk factor

It should be noted that the overall system risk factor depends on the execution probabilities of the terminal use cases. In other words, terminal use cases which have low execution probability, such as for example the Mode_Setting use case in our case study, typically will not contribute much to the overall system risk factor. From this perspective, system analysts very often will need to take into account use cases with high risk factors, regardless of their execution probability and their contribution towards the system risk factor.

5.6 Summary and Discussion

As the software systems become more complicated, there are more composite functional dependencies within the system. Considering object-oriented software models, these functional dependencies are captured in terms of use case relationships. In any analysis of UML models of software systems that deals with use cases, it would not be acceptable to overlook these relationships among use-cases and treat them as independent. In this chapter, we addressed UML use case relationships for quantitative analysis of reliability-based risk assessment.

5. Reliability-Based Risk Assessment with Use Case Relationships

We presented a new algorithm for risk assessment that generalizes our earlier work on risk assessment by relaxing the assumption that use cases are independent. For this purpose, we first proposed a method which is used to estimate the risk factor of a non-primitive use case related to a primitive use case by either <<extend>> or <<include>> relationship. Then, we proposed an algorithm that allows us to estimate the use cases and system risk factors from a general use case diagram that may include many different <<extend>> and <<include>> relationships, the most widely used relationships between use cases. Finally, we applied the generalized methodology on an industrial case study.

In order to validate the results obtained from the proposed methodology against other sources of risk evaluation, we are focusing on applying the generalized methodology on other case studies that have reliability-based risk assessment data for their components.

6 Change Propagation Metrics

In the next chapter, we address maintainability-based risk assessment which tries to assess how difficult in the future the system maintenance will be due to current maintenance tasks. This assessment methodology relies on change propagation probabilities and the size of change between components of the architecture. Thus, in this chapter we define and provide an estimate of the probabilities of changes that arises in a component (in the context of corrective/adaptive/perfective maintenance) requiring changes to other components and their corresponding size of change in these components. We introduce, analyze, and validate formulas for estimating these probabilities using architectural level information.

6.1 Change Propagation Probabilities

Let us consider a software architecture modeled by components and connectors. We are interested in the maintainability of the products instantiated from it. In corrective or perfective maintenance tasks, *change propagation probability* matrix for an architecture reflects on the probability of changing component C_j as a result of a change to component C_i.[Abdelmoez+ 2005A]. The estimation of the elements cp_{ij} of the change propagation matrix CP is based on the following definition:

Definition 1.

Given components C_i and C_j of a system *S*, the *change propagation probability* from C_i to C_j is denoted by cp_{ij} and defined as the following conditional probability

$$cp_{ij} = \Pr(([C_j]\neq[C_j'])|([C_i]\neq[C_i'])\wedge([S]=[S'])), \qquad (6.1)$$

where [X] denotes the functionality of component/system X and *S'* is the system obtained from *S* by changing C_i into C_i' (and possibly C_j into C_j' as a consequence).

In practice it is useful to add some qualifications to the above definition and distinguish between the *1-step* and *multi-step change propagation*. The *1-step change propagation,* accounts for the change propagating from one component to another directly as a result of one component using services (information) provided by another, i.e., "in one step". We denote the *1-step change propagation* by CP_1.The term *n-step change propagation* ($n\geq 2$) refers to the probability of a change propagating from one component to another as a result of *n* consecutive acts of 1-step change propagation. We denote the *n-step change propagation* by CP_n

6.1.1 Change Propagation Usage

We submit that the matrix of change propagation probabilities can help the software architect to make assessments and take decisions regarding the projected maintainability of software products that stem from the architecture at hand. Specifically, we cite the following uses:

- A summary inspection of the matrix can reveal quickly the difficulty and the cost of maintenance operations on the system. An idealistic (and usually uninteresting) system that is perfectly modular has an identity matrix, whereby each change is localized to the component where it is applied and does not propagate to other components (in principle, this is possible only if the components are independent); the case of unrelated components notwithstanding, the closer a matrix is to the identity, the better.
- If the row corresponding to a component *A* has high values, we infer that changes to this component must be avoided because they propagate widely throughout the system. Preventive measures include focusing verification and validation activity on this component (to minimize subsequent corrective maintenance), and optimizing the design of this component (to minimize subsequent perfective maintenance).
- If the column corresponding to a component A has high values, we infer that this component is likely to undergo frequent changes in the maintenance phase. Preventive measures include special care to design this component for ease of modification.
- The matrix of change propagation probabilities can also be used to compare candidate system architectures when maintenance costs are an important consideration. To this effect, we have Change Propagation Coefficient which is a quantitative measure of diagonality, which indicates to what extent a given matrix is diagonal (i.e. has non-zero values in the diagonal and near zero values outside the diagonal). This measure can be used to give a summary comparison of candidate architectures when other, more meaningful, criteria fail to discriminate between the candidates.

6.1.2 Analytical estimates of Change Propagation Probabilities

The purpose of the analytical step is to derive a formula for estimating change propagation probabilities using architectural information. For the sake of tractability, we alter the definition slightly, prior to our analytical study:

$$cp_{ij} = \Pr(([C_j]\neq[\ C_j'])|([C_i]\neq[C_i']) \wedge ([\ C_i * C_j]=[\ C_i' * C_j'])) \qquad (6.2)$$

In other words, the probability that we are computing is not conditioned on the overall system function being preserved ([S]=[S']) but rather on the aggregate composition of C_i by C_j being preserved ([C_i * C_j]=[C_i' * C_j']), where * is an operation that reflects the interaction between C_i and C_j in system S. We find it useful to distinguish, conceptually, between two broad classes of changes in C_i:

- The case where the interface between C_i and C_j remains unchanged but the function of C_i changes.
- The case where the interface between C_i and C_j changes.

Also, we observe empirically, and can easily verify analytically (by a combinatorial argument), that the probability of a change propagation under the first condition (preserving the interface, altering the actual function) is very low: there are very few changes in the function of C_i that we can make without having to change C_j as a consequence. Factoring these observations into the revised formula of change propagation, we find that we can approximate the change propagation probability from C_i and C_j by the formula:

$$cp_{ij} = \Pr(([C_j]\neq[C_j'])|(I_{Ci,Cj} \neq I_{Ci',Cj'}) \wedge ([C_i * C_j]=[C_i' * C_j'])) \qquad (6.3)$$

where $I_{Ci,Cj}$ is the interface (i.e. set of relevant connectors) between C_i and C_j.

An architecture can be seen as a collection of components C_i, i=1,…,N. With every component C_i, we associate the set V_i of the interface elements of the provided functions of C_i. We determine the *usage coefficient* value π_v^{ij} for every interface element $v \in V_i$ and every other component C_j, $j \neq i$. They take binary values:

- π_v^{ij} =1, if the interface element v provided by C_i is required by C_j. This means that any signature change in component C_i associated with interface element v will propagate to component C_j.
- π_v^{ij} =0, otherwise.

Hence, for every pair of components C_i and C_j, i≠j, the change propagation probability cp_{ij} can be estimated based on the values of the *usage coefficients* π_v^{ij} by:

$$cp_{ij} = \frac{1}{|V_i|} \sum_{v \in V_i} \pi_v^{ij} , \qquad (6.4)$$

We need to remark here that formula (6.4) is based on the assumption that an interface in C_i whose propagation we are trying to trace is equally likely to affect any of its interface elements, as shown in Figure 33.

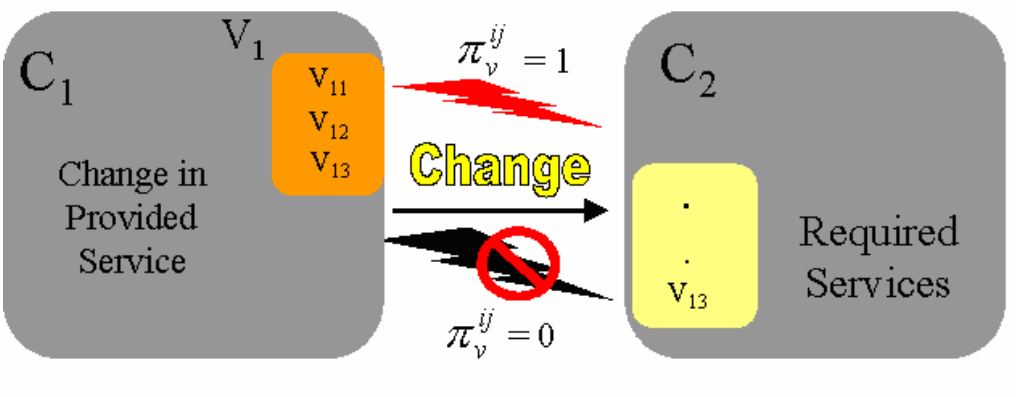


Figure 33 Single-step change propagation estimation

The method described above allows us to evaluate the (1-step) change propagation probabilities, given the information on the interface specification of the architecture. Once the 1-step CP values are obtained, we can get upper-bound estimates on the multi-step change propagation.

6.1.3 Multi Step Change Propagation

Suppose we have obtained the 1-step change propagation matrix CP using (6.4). For convenience, here we use shorter notations:

$$\Lambda(i, j) := \mathrm{CP}(C_i, C_j), \qquad i, j = 1,\ldots,N \qquad (6.5)$$

In the directed graph G representing our architecture, let $\Pi_G(i, j)$ be the set of all simple (directed) paths leading from node i (i.e. component C_i) to node j ($i \neq j$). For a path $\pi = (i, i_1, i_2,\ldots, i_{n-1}, j) \in \Pi_G(i, j)$, where $n = |\pi|$ is the length of π, let us denote by $\Lambda(\pi)$ the probability of a change in i propagating to j via the path π, i.e., the probability that a change in i causes a change in i_1, which in turn causes a change in i_2, etc., finally causing a change in j. If we make the simplifying assumption that change propagation events in different connectors are independent, we obtain:

$$\Lambda(\pi) = \Lambda(i, i_1)\, \Lambda(i_1, i_2) \ldots \Lambda(i_{n-1}, j). \qquad (6.6)$$

Let $\Lambda_k(i, j)$ be the probability of a change in i propagating to j in k steps ($k \geq 1$). It is easy to see that $\Lambda_k(i, j)$ is the probability of the union of the events that consist in the change propagating

from *i* to *j* along particular simple paths of length *k* in *G*. Since the probability of a union is never greater than the sum of the probabilities of the constituent events, we have:

$$\Lambda_k(i, j) \leq \sum_{\substack{\pi \in \Pi_G(i, j) \\ |\pi| = k}} \Lambda(\pi) \qquad (6.7)$$

The reason we only consider *simple* paths is that we are interested only in the *first* propagation of a change to its destination component. From inequality (6.7) we can obtain the following result that supplies an easily computable upper bound for the n-step change propagation.

Proposition 1:

$$\Lambda_n(i, j) \leq \Lambda^n(i, j)\ , \qquad (6.8)$$

where Λ^n is just the n^{th} power of the change propagation matrix Λ.

This, in particular, means that even though we can multiply CP probabilities along any particular path (without making any independence assumptions), we cannot sum these products for various paths, to get an n-step CP value as a result. In other words, the n-step CP matrix cannot be obtained by merely taking n^{th} power of the original 1-step CP matrix. The n^{th} power give us an upper bound for the n-step CP probabilities.

6.2 Predicting Change Propagation Patterns

In addition to the many applications we have briefly discussed, we find that the availability of change propagation probabilities of an architecture allows us to quantify an important classification of change propagations, first proposed by Clarkson et al. [Clarkson+ 2000] ,Clarkson et al present a three-tiered classification of changes as follows:

1. *Ripple of change:* the introduced changes will result in an acceptable behavior to be observed for the maintenance process. Changes are controlled and limited.
2. *Wave of change:* the introduced changes will still result in an acceptable behavior to be observed for the maintenance process. Although there are many changes, they are under control.

3. *Avalanches of changes:* the introduced changes will result in an unacceptable behavior to be observed in the maintenance process. There are many changes and they are uncontrolled.

The avalanche type of change propagation is the one that software maintainers worry most about; such changes would make managing the software maintenance very difficult and very costly. It would be a great benefit to be able to predict in advance if a certain component is prone to this type of change propagation.

Because of its highly heuristic and qualitative nature, the above classification [Clarkson+ 2000], while being conceptually useful, cannot be applied directly for an analytical study. In order to make *Clarkson's classification* more usable for the purposes of our quantitative analysis of change propagation, we give it a more rigorous quantitative interpretation. Whereas Clarkson et al present this classification to characterize individual changes; we use it to characterize components. Specifically,

- We consider that a component belongs to the *Ripple* class if, on average, the changes initiated in this component produce a ripple effect.
- We consider that a component belongs to the *Wave* class if, on average, the changes initiated in this component produce a wave effect.
- We consider that a component belongs to the *Avalanche* class if, on average, the changes initiated in this component produce an avalanche effect.

In order to give formalization to these concepts, we must introduce some numeric parameters.

- The *negligibility threshold* δ ($0< \delta<1$), indicates the level below which the change propagation probability is considered negligible.
- The *propagation area significance* α ($0 < \alpha < 1$), indicates the fraction of the total number of the system components affected by a single component that can be considered significant in a single step.
- The *ripple threshold* τ ($0< \tau<1$), determines the fraction of the total number of components affected by ripple change propagation.
- The *avalanche threshold* ψ ($0< \psi<1$), determines the fraction of the total number of components that must be affected in order for a change to be considered an avalanche.

Having chosen a value of δ, we can, for each integer $n>0$ define the n-th step CP-graph $CPG_{n,\delta}$ of the architecture to be the subgraph of the original architecture graph G obtained by

erasing in G all the edges (A,B) for which $CP_n(A,B) < \delta$. Notice that the graph $CPG_{n,\delta}$ monotonically decreases as δ increases (for δ sufficiently close to 0, $CPG_{n,\delta}$ = G, while for δ sufficiently close to 1, it is empty).

Definition 2.

The *n-th step CP range* of A (with sensitivity threshold δ), denoted by $M_{n,\delta}(A)$, is the out-degree of the node A in the graph $CPG_{n,\delta}$, i.e.,

$$M_{n,\delta}(A) = |\{B \in S | CP_n(A,B) > \delta\}| \qquad (6.9)$$

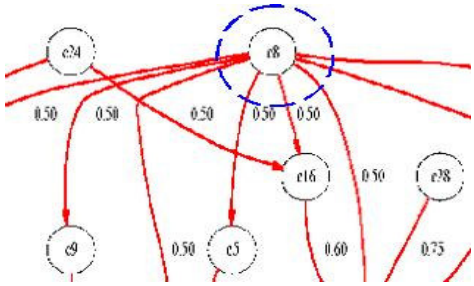


Figure 34 An example on how to calculate $Mn(C_8) = 8$

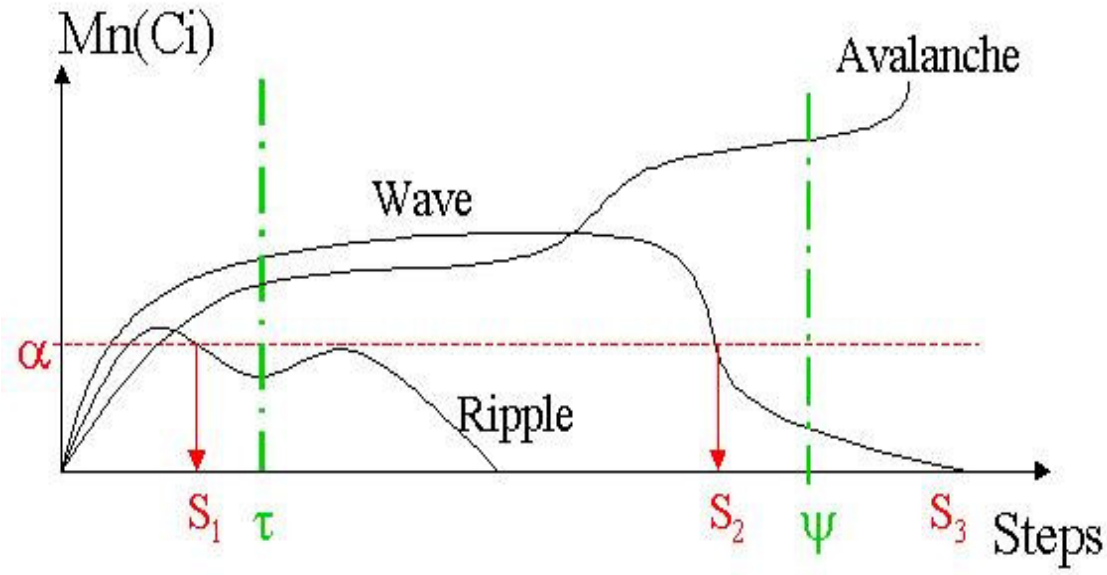
If we apply the definition on the part of the graph for a single-step change propagation for component C_8 presented in Figure 34 , we find that $Mn(C_8) = 8$. Based on these metrics, we can interpret Clarkson's classification of CP behavioral patterns of the system according to the dynamics of $M_{n,\delta}(A)$ considered as a function of the step n. (Here, we interpret the step of the unfolding process of change propagation, as an analogue of the time into the maintenance cycle in Clarkson's classification.)

Definition 3.

For any component A in the architecture S, we say that A has a potential for generating

- *a ripple of changes, if* $M_{n,\delta}(A) < \alpha|S|$ *for* $n > \tau|S|$ *, i.e., steps of change emanating from A beyond the ripple threshold* $\tau|S|$ *have negligible effect on any other component;*
- *an avalanche of changes, if* $M_{n_a,\delta}(A) \geq \alpha|S|$ *for* $n \geq \psi|S|$ *, i.e., steps of change emanating from A beyond the avalanche threshold* $\psi|S|$ *affects more than* $\alpha|S|$ *of the components;*
- *a wave of changes, if neither of the two conditions above are satisfied (i.e., it is neither ripple nor avalanche).*

An illustrative description for change propagation behavior is presented in Figure 35. Using these parameters, we can now characterize Clarkson's classification using change propagation probabilities.



Where:

α *(0 <* α *< 1) is the propagation area significance,*

τ *(0 <* τ *< 1) is the ripple threshold, and*

ψ*(0 <* ψ *< 1) is the avalanche threshold*

Figure 35 Parameterization of the categorization of the change behavior

Furthermore, we attempt to develop some heuristic methods of predicting the type of change propagation behavior expected to occur in the system (ripple, wave, or avalanche). It is easy to see that for the n-step change CP probability from component A to component B to be positive, it is necessary that there exist in the architecture graph G a simple path starting at A and ending at B of length exactly n. From this observation we derive the following proposition, which we present without formal proof.

Proposition 2:

Let the directed graph $G^{(n)}$ *be derived from the architecture graph G by the following rule: there is an edge from A to B in* $G^{(n)}$ *if and only if there exists in G a directed path of length n from A to B. Then the n-th step CP-graph* $CPG_{n,\delta}$ *(defined in Section 2) for any* δ *is a sub-*

graph of $G^{(n)}$. In particular, the n-th step CP range of A (with any sensitivity threshold δ) $M_{n,\delta}(A)$ is never greater than the out-degree of A in graph $G^{(n)}$.

Proposition 2 can be used to classify some components as being capable of producing only a *ripple change* (as defined in Definition 3) without even estimating the CP probabilities between it and other components. Namely, we have the following proposition.

Proposition 3:

If the out-degree of node A is less than $\rho |S|$ in the architecture graph G and in $G^{(1)}$, and is zero (no outgoing edges) in $G^{(n)}$ for all $n > 2$, then component A generates only ripple changes.

Similarly, one may be able to conclude simply from the topological structure of the architecture graph that a certain component cannot produce an avalanche change.

6.3 Experimental Change Propagation

In this section, we discuss an experiment that we ran to validate the analytical formula that we propose for estimating change propagation. First, we present a brief description of the sample system that we chose for this experiment. Then, we apply the aforementioned single-step change propagation on the system. We also use a controlled experiment of *"mutation operators"* changes to see how well the analytical results correlate with the controlled experiment results of the system under investigation.

The system we have selected for our experiment is a spreadsheet application written in Java, named Sharp Tools. The details of the Sharp Tools case study are presented in Appendix I.D.1. Using the interface specification of the system, we first determine the interface elements that have an effect on the neighboring components of the architecture of the system. Then, we analytically compute an estimated change propagation matrix for the system. This gives us an estimate of the probability that an interface change will propagate to a neighboring component due to that change.

To have a better understanding of the resulting matrix, we produce a graphical representation of it. In Figure 36, we present the critical change propagation of the system, using a high threshold of significance; specifically, we let the significance threshold be 0.4 in order to identify the more critical components of the system.

It is worth noting that this graph cannot be considered as a Markov chain model, as the sum of the probabilities outgoing from a node can go beyond 1, violating the Markov constrain. For example if we examine component C8, we find the sum of probabilities of a change propagating from C8 to other components is greater than 1.

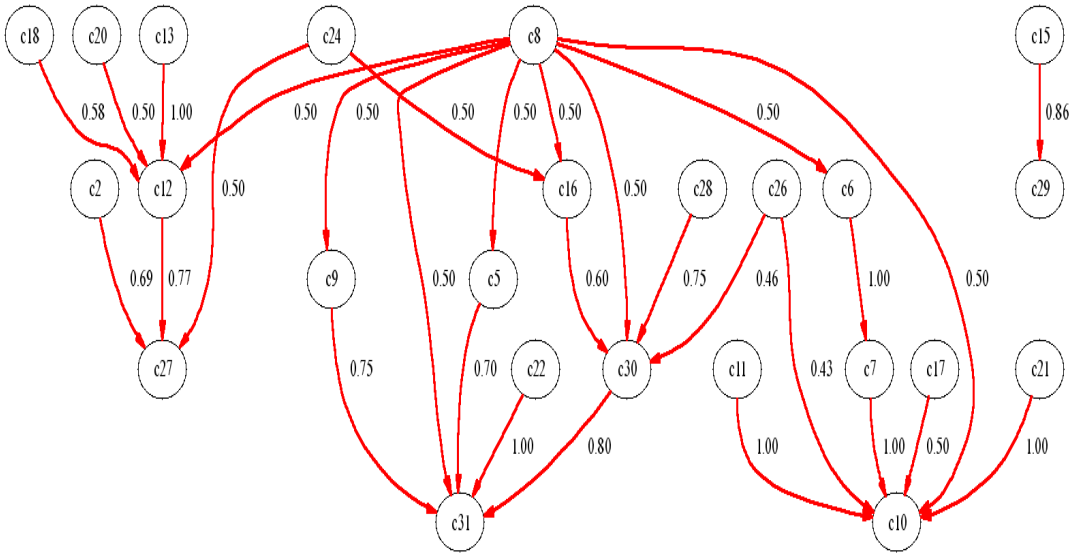


Figure 36 Graphical representation of the critical change propagation

In the *"mutation operators"* controlled experiment, we introduce changes into interface element of the components of the Sharp Tools case study. Specifically, we change the variable types of the methods signature in each interface element in each component. With the help of a compiler, we determine the components that are affected by the changes introduced. The ratio of the times the component required to be changed and the times of changes introduced into that component interface is an estimate of the empirical change propagation probability.

It is worth mentioning that *"mutation operators"* analysis is time and effort consuming. In the following sections, we compare the analytical results with the controlled experiment results of the Sharp Tools case study. Then, we expand our single step change propagation results to get a multi-step change propagation view of the same system. Using these results, we to assess the change propagation behavior of this system

6.4 Change Propagation Probabilities Validation

In this section, we judge the results evaluated by the analytical formula against the results derived from the controlled experiment of *"mutation operators"* to assess the validity of our analytical formulas.

6.4.1 Correlating Single Step Change Propagation Matrices

In this section, we present the results of the study that we conducted to explore the correlation between the analytically estimated single-step change propagation matrix C_A and its experimentally derived counterpart C_E. The correlation coefficient between all the cells of the analytical single-step matrix and the experimental single-step matrix is:

$$Cor(C_A,C_E) = 0.93 \quad \text{(r value)} \qquad (6.10)$$

$$R\text{-}Sq = 0.8649,$$

Where "r" denotes the Pearson product-moment correlation coefficient.

For the nontrivial values (other than those that are either 0 or 1 by definition), the rationale behind this criterion is that trivial values do not really test our analytical results, we find:

$$Cor(C_A,C_E) = 0.85 \quad \text{(r value)} \qquad (6.11)$$

$$R\text{-}Sq = 0.7225$$

A *significant* relationship between some variables does not necessarily mean that the relationship is very useful in building predictive models. Thus the R-Sq values are also shown above to assess the explanatory power of each model.

6.4.2 Statistical Significance of the Correlations

Now we need to validate our correlation results, i.e., to make sure that the positive correlation values we observed are statistically significant (did not occur by chance). To test the relationship between analytical and experimental change propagation, a statistical hypothesis testing was performed using the *t*-test (one-tail) for the nontrivial entries using the level of significance $\alpha = 0.05$.

Our hypotheses were

- *H0* : ρ = 0 (There is no linear association between analytical change propagation values and empirical error propagation values)
- *H1* : ρ > 0 (There is a positive linear association between analytical change propagation values and empirical error propagation values)

where ρ denotes the correlation coefficient.

We have computed the value of the *t* statistic for the nontrivial values of single-step matrices as t_{ob} = 18.98632 (with n=140 samples), and the corresponding P-value is less than α = 0.05. Thus, we reject the null hypothesis of no correlation, and thus infer that the correlation of 0.85 is statistically significant. The *t*-test results showed that the fairly high correlation values between analytical and experimental change propagation values we obtained did not occur by chance (i.e., are statistically significant).

6.5 Multi-Step Change Propagation Matrix

Form the single-step change propagation results, we can get an estimate of the multi-step change propagation according to Section 6.1.3.We can then estimate the outgoing change propagation of a component as the total change propagation that is exported by this component to other components. We can track the outgoing change propagation as change propagates in multi-steps, and observe the behavior of this component. Thus, we can categorize the components according to their outgoing change propagation as ripple, wave or avalanche.

Ripple components are those having outgoing change propagation that dies out rapidly with very few steps of change propagation. Wave components are those that sustain a large value of outgoing change propagation for a number of steps, but this value dies out eventually. Avalanche components are those that have an increasing value of outgoing change propagation as the number of steps of change propagation increase, and this value does not die out eventually. From the experimental results, we can recognize three patterns of outgoing change propagation for ripple, wave and avalanche components.

For each component in the architecture, The n-th step CP range Mn(Ci), which is directly proportional to outgoing change propagation, is shown in Figure 37. We can see that there are only ripple and wave components and no avalanche components. So, we can expect that, when making a change in this system, we can recognize a ripple change propagation or at most a wave of change propagation. But, it is highly unlikely to have an avalanche change.

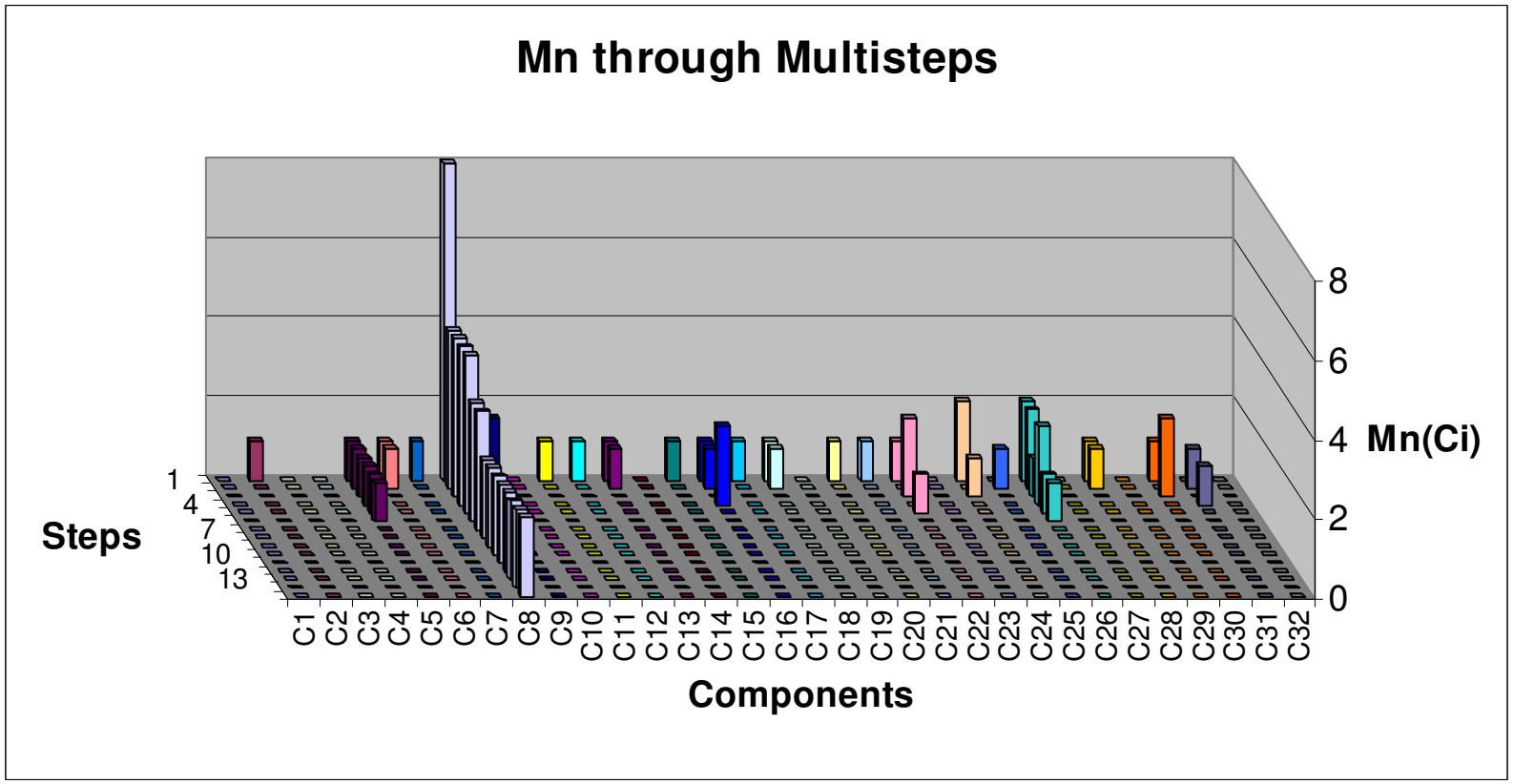


Figure 37 Mn(Ci) of the components through multi-step change propagation

We can recognize that any change for component C8 should be handled with care, as it is a component with high change propagating probabilities. Any maintenance effort that may be needed to deal with this component should be expected to cause a wave of changes since this is a highly centralized component that might affect the others much by its high mutual dependencies. Figure 38 shows a pattern for ripple components where the Mn(Ci) tends to decay in very few steps. When checking Figure 39, we find a pattern of a potential avalanche component. The Mn(Ci) still have a significant magnitude over a large number of steps. In Figure 40, we can recognize a pattern of wave components that are midway between the ripple and the avalanche components.

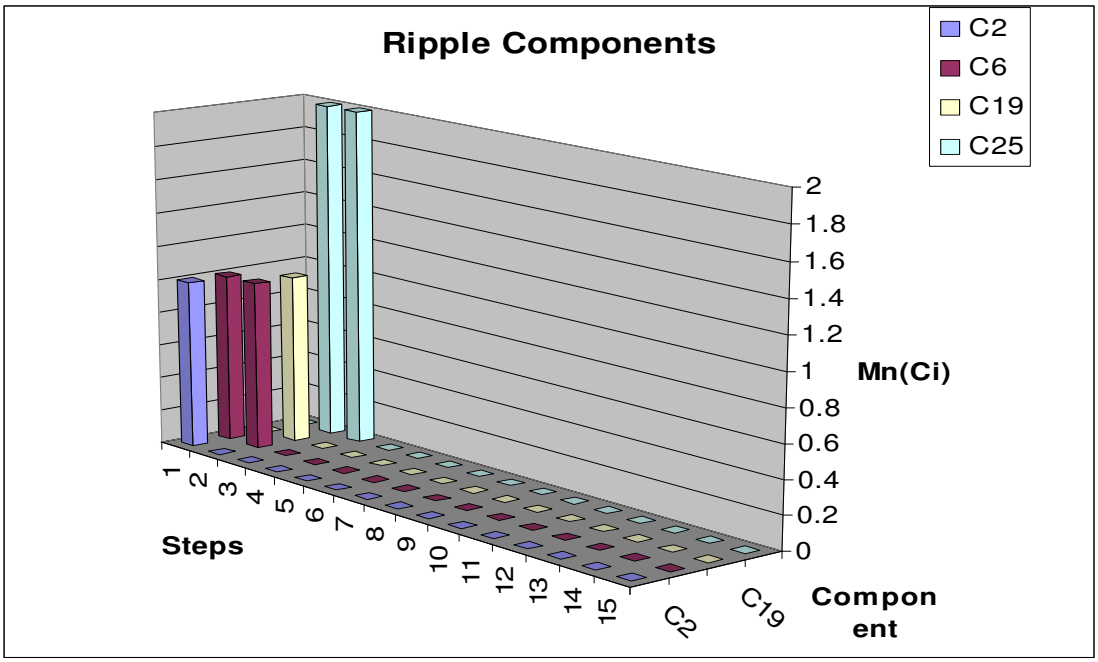


Figure 38 Pattern of Ripple components

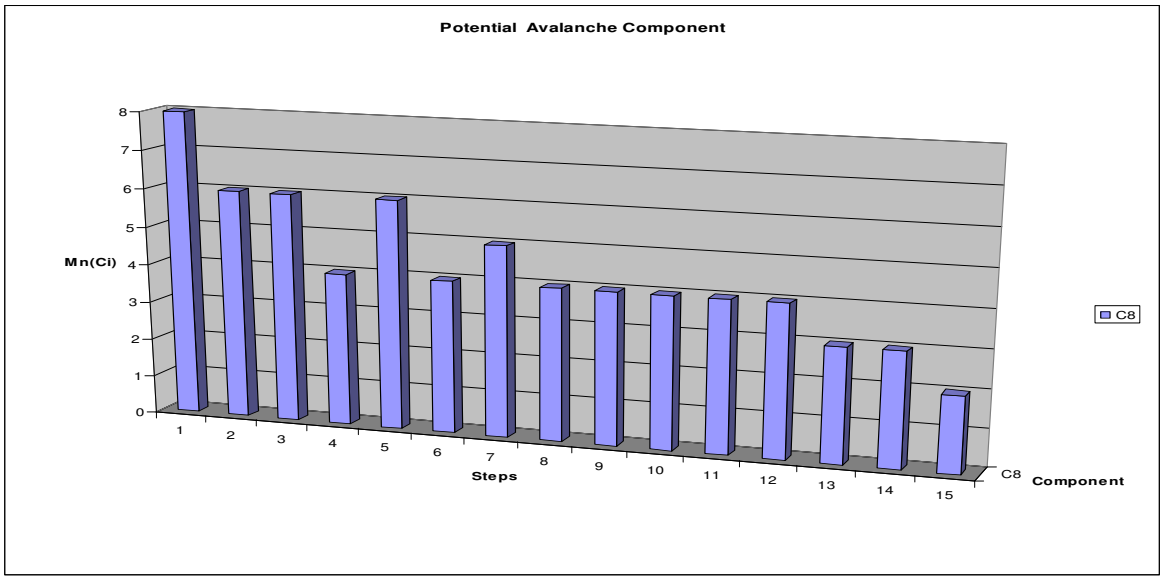


Figure 39 Pattern of a potential Avalanche component

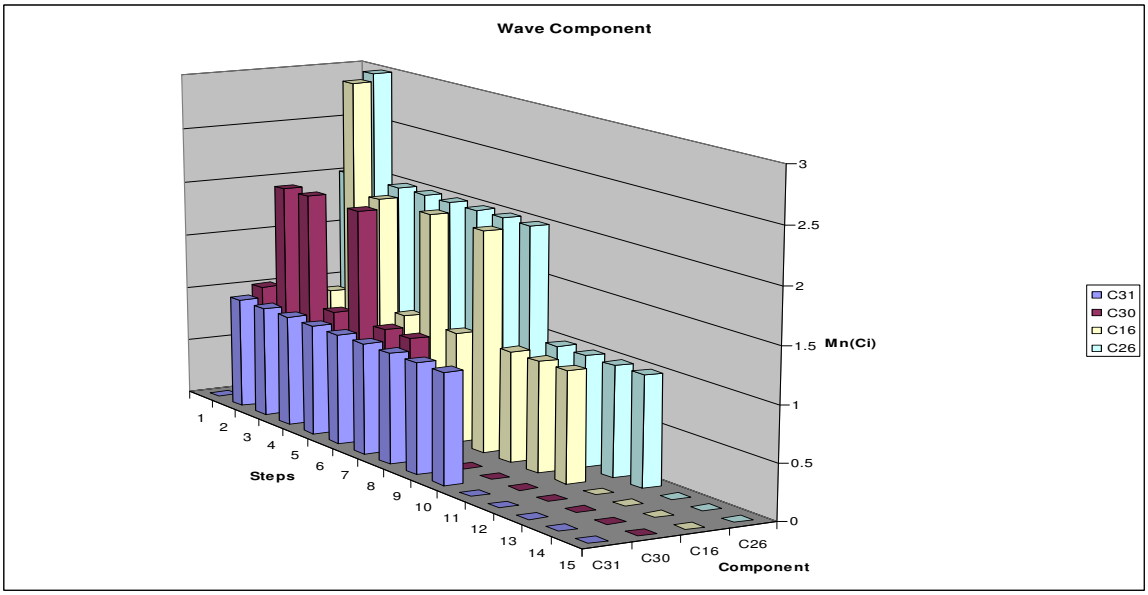


Figure 40 Pattern of Wave components

6.6 Using Change Propagation Probabilities to Assess Quality Attributes of Software Architectures

In [Shaik 2006], we have designed an experiment that compares architectures using object-oriented metrics and change propagation matrices. The goal of the experiment is to assess to what extent the object-oriented metrics on one hand and the Change propagation probability matrices on the other hand are good predictors of architectural quality attributes. To this effect, we consider sample applications, and

derive two candidate architectures for each: one that is based on design patterns (hence is presumably of higher quality) and one that is design ad-hoc, without predefined patterns. The following methodology has been applied:

1. Prepare a pair of architectures for the same application. One of the architectures is designed using design patterns while the other has no patterns.
2. Apply the CP metric on both architectures.
3. Apply other object-oriented metrics on both architectures.
4. Analyze and compare the results. The architecture the employs software patterns should have a better quality in terms of extensibility and maintainability.

It is worth noting that we carry out the comparison of design quality for the architectures with the help of the Software Architecture Change Propagation Tool (SACPT) [Abdelmoez+ 2004B]. Inputs to our tool are obtained with the help of Understand for Java tool [JavaUnderstand]. SACPT generates the CP matrix of components in the architecture.

Our first example is a simple application where an employer is seeking employment applications for the various jobs available, which are submitted through detailed electronic forms that must be validated. There are two versions; one version is a simple switch case whereas the other version is implemented using the strategy pattern.(See details in Appendix I.D.2). We restrict the analysis to the components that exist before and after the application of the pattern.

Figure 41 shows the change propagation probabilities of the case study when using switch cases. Figure 42 shows the change propagation probabilities of the case study after applying the strategy design pattern. The Change Propagation Coefficient (CPC) for the architectures before and after using the design pattern are 0.18 and 0.11 respectively. From the CPC values, the architecture, which employs a design pattern, is better in design quality than the one which does not. Figure 43 shows the Weighted Methods per Class (WMC) and McCabe Cyclomatic Complexity metrics (MCC) for the components before and after applying the pattern. The JobApplicantForm component has been improved by employing strategy design pattern.

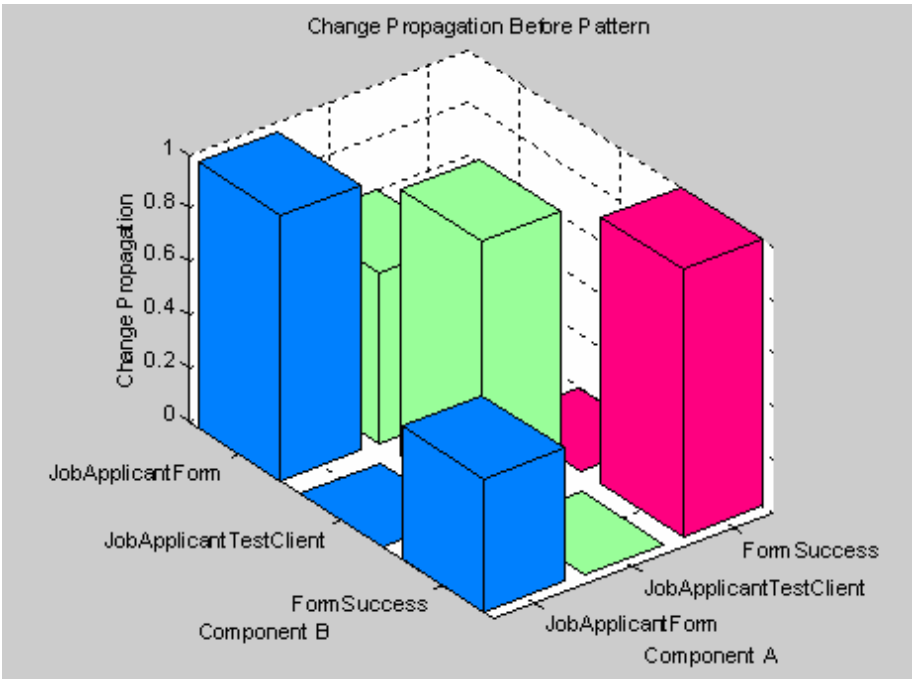


Figure 41 Change propagation of Job Application before applying strategy pattern.

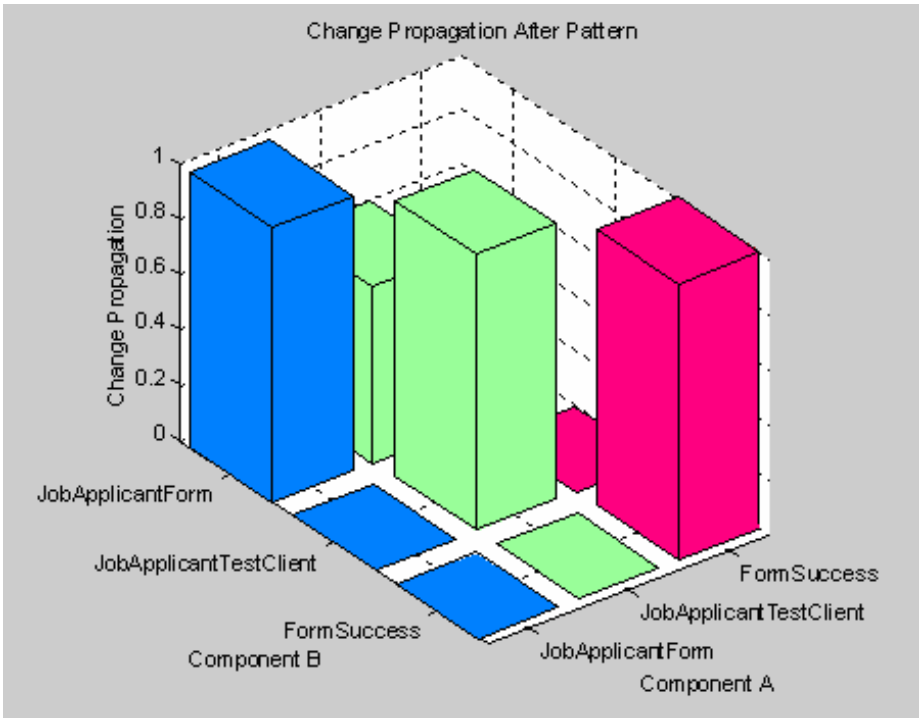


Figure 42 Change propagation of Job Application after applying strategy pattern.

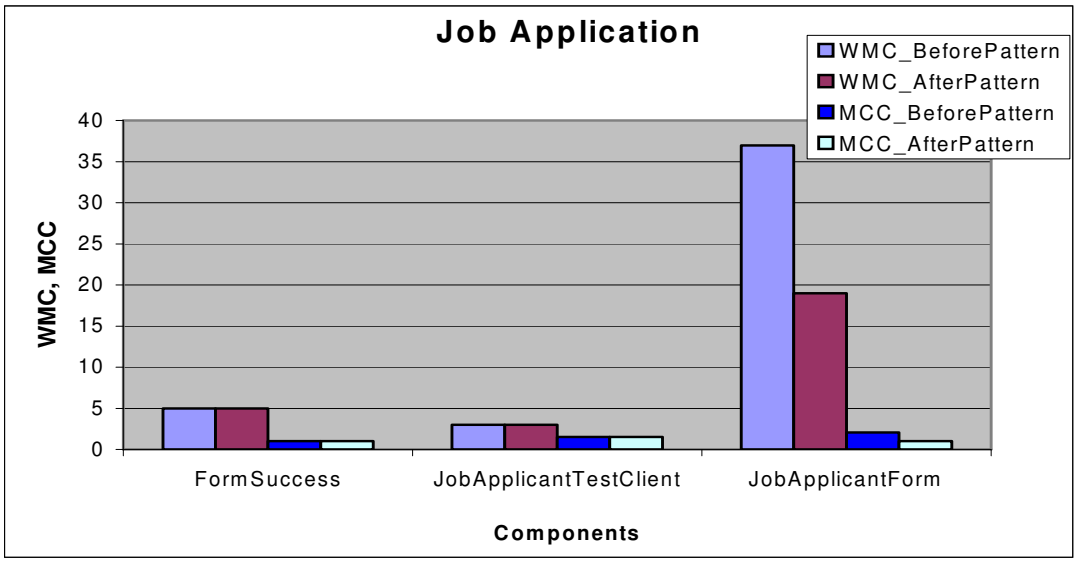


Figure 43 Weighted Methods per Class and McCabe Cyclomatic Complexity for Job Application

The second example is an application that tracks the states of colleague components. Each colleague will update its state according to its current state and the changes to the states of the other colleagues. (See details in Appendix I.D.3). Figure 44 and Figure 45 show the change propagation probabilities of the architectures before and after applying the mediator pattern. In Figure 44, we recognize that the three colleague components are tightly coupled to one another. In Figure 45, adding the mediator pattern decreased coupling between the colleague components. The three colleague components are completely decoupled with respect to one another. The Change Propagation Coefficient CPC for the architecture that does not employ any design pattern is 0.11, where as CPC value for the one, which employs mediator design pattern is 0.05. From the CPC values, the architecture, which employs a design pattern, is better in design quality when compared to the same architecture that does not.

Figure 46 shows the Weighted Methods per Class (WMC) and McCabe Cyclomatic Complexity (MCC) metrics for the architecture before and after using the mediator pattern. All the three colleague components have been improved in terms of complexity when a mediator design pattern has been employed. Both the metrics WMC and MCC show this improvement as a decrease in value for the components in the architecture that employs a design pattern.

One of the disadvantages of using the mediator design pattern is the localization of the behavior of the components. The distributed behavior of the components is localized to few components in the architecture that employs a mediator design pattern. Even though, the architecture that employs a mediator design pattern is more extensible, maintainable and reusable than the one which does not. When

adding new colleagues to the architecture, we need only to change the components of mediator pattern. All the other colleagues will not be affected as they will not be directly coupled to the new colleague component.

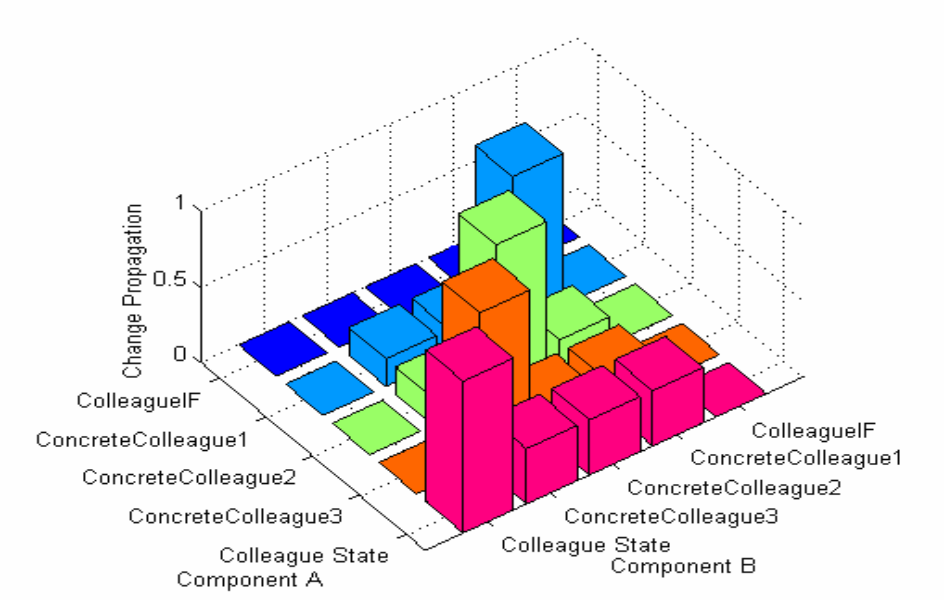


Figure 44 Change propagation probabilities for the simple design on case study Colleague States

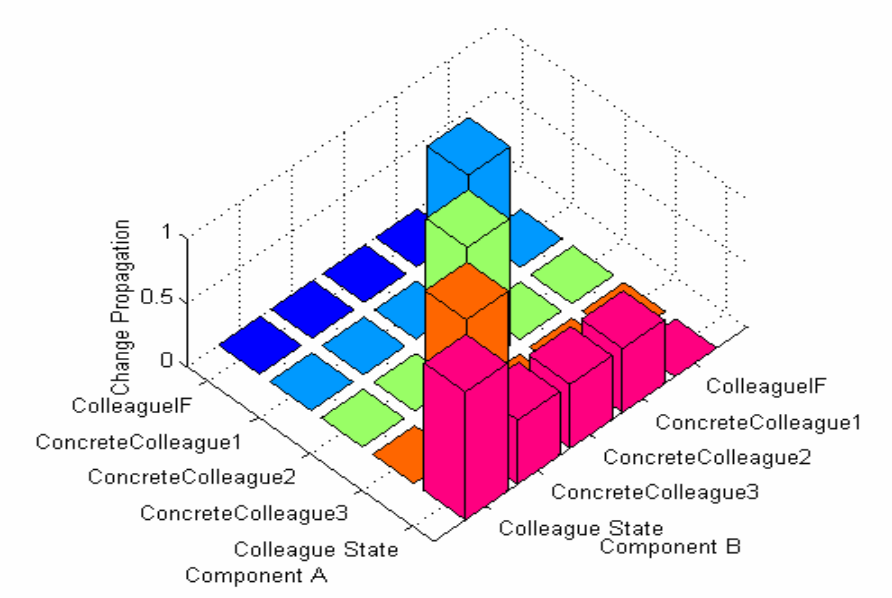


Figure 45 Change propagation probabilities for the architecture employing mediator design pattern

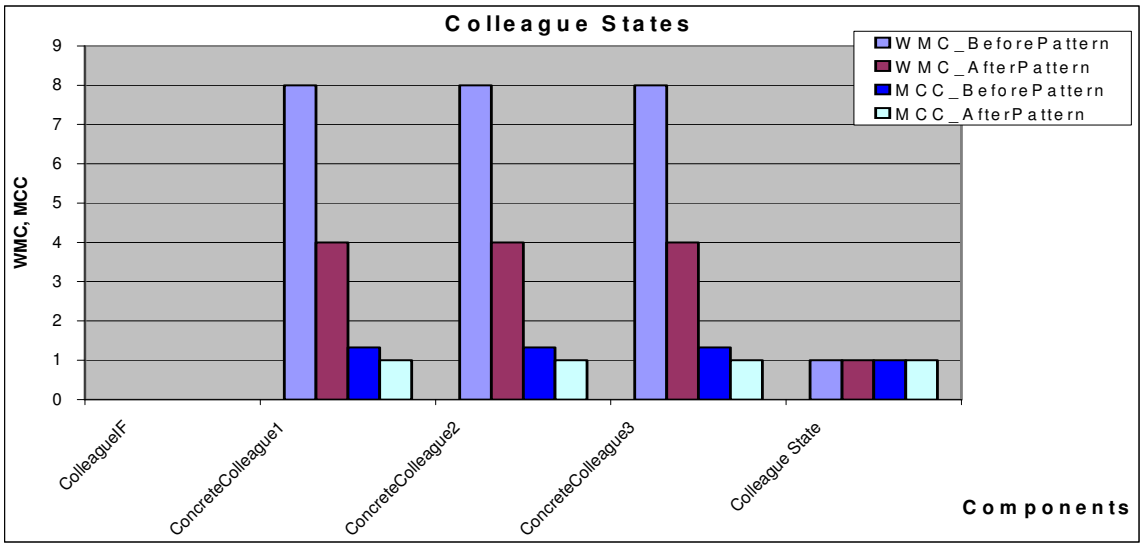


Figure 46 Weighted Methods per Class and McCabe Cyclomatic Complexity for Colleague States

6.6.1 Comparison of Change Propagation Metric with Other Metrics

In this section, the change propagation metric is compared with respect to three other coupling-based, object-oriented metrics: Coupling Between Objects (CBO), Response For a Class (RFC) and Message Passing Coupling (MPC). We restrict the analysis to the components that exist before and after the application of the pattern. Figure 47 shows the Coupling Between Objects (CBO) for the two case studies before and after the application of the design pattern. Observing Figure 47, one can conclude that CBO metric is not sufficient to compare and state that an architecture is better in design quality when compared with another. Though there are improvements in the CBO values for the three colleague components in the Colleague States case study, there is an increase in the CBO value for the JobApplicationForm component in the architectures which implements the strategy pattern for case study Job Application.

In Figure 48, the Response For a Class (RFC) metric could not show the difference between the two architectures in the Job Application case study. In fact, it showed that the architecture that employs a simple design is better in design quality than the architecture that employs strategy design pattern. On the other hand, for the Colleague States case study the RFC metric confirmed a relative improvement of using design patterns. This shows that RFC metric may not be a good choice to compare between two candidate architectures. Furthermore, the MPC metric showed no difference in design when applied to both the architectures on each of the two case studies. Figure 49 shows the values for the MPC metric computed on the Job Application and the Colleague States case studies. MPC metric could not show the difference between the architectures in both cases.

6. Change Propagation Metrics

On the other hand the change propagation probability metric shows these variations and points out that one of the architectures is better than the other. Check the change propagation probability values for the three colleagues components in the Colleague States case study, Figure 41 and Figure 42, and JobApplicationForm component in the Job Application case study, Figure 44 and Figure 45. Change propagation probability metric can show different perspective and complements the usage of other object oriented metrics: CBO, MPC and RFC in both case studies.

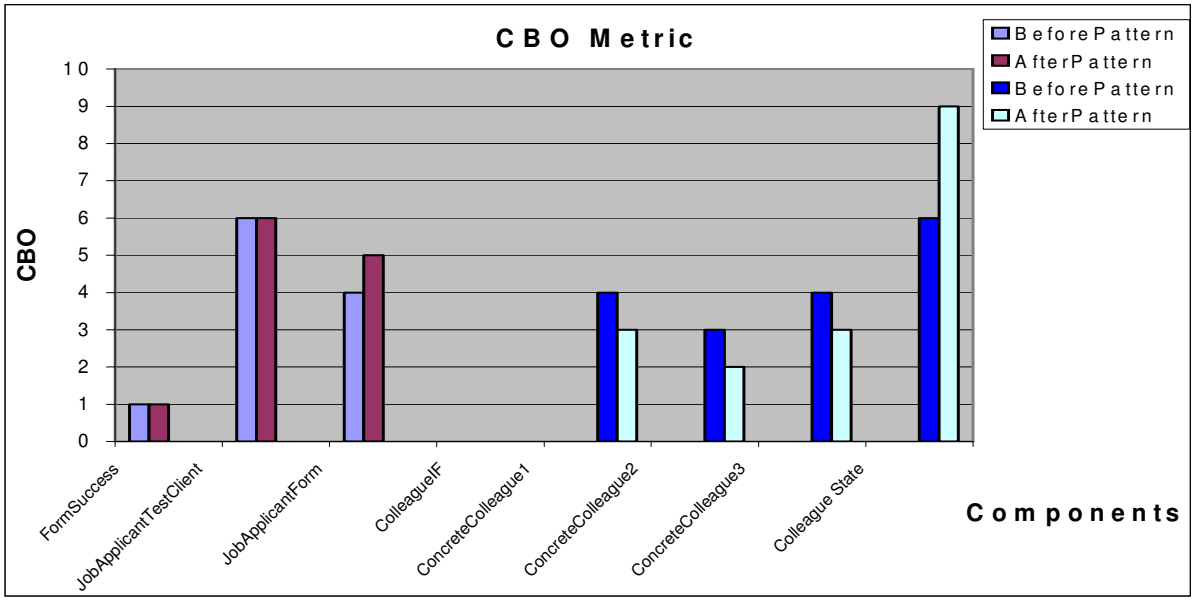


Figure 47 CBO for the case studies on Colleague States and Job Application

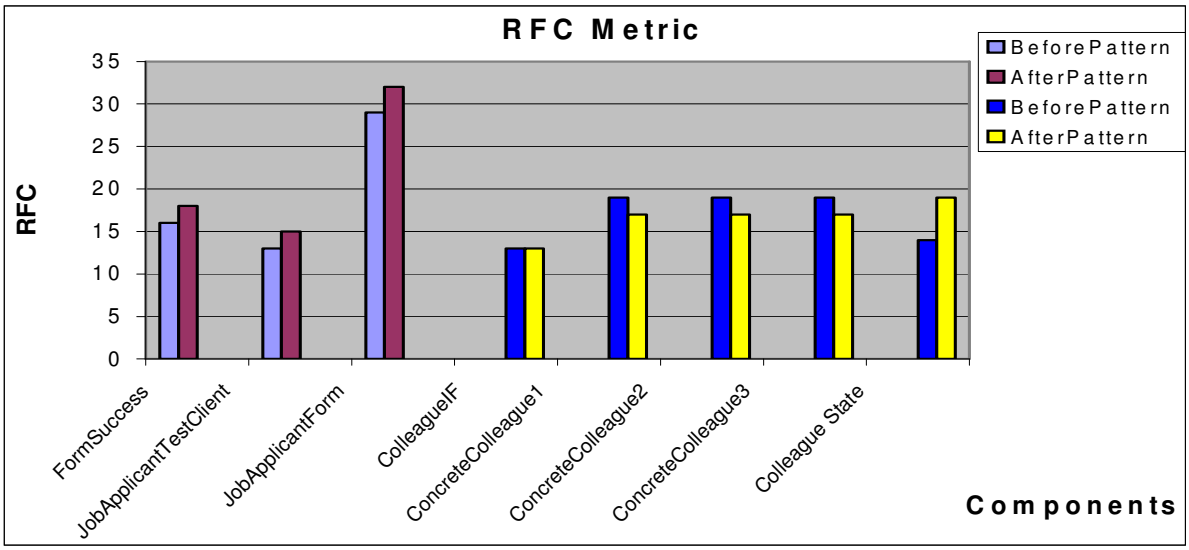


Figure 48 RFC for the case studies Colleague States and Job Application

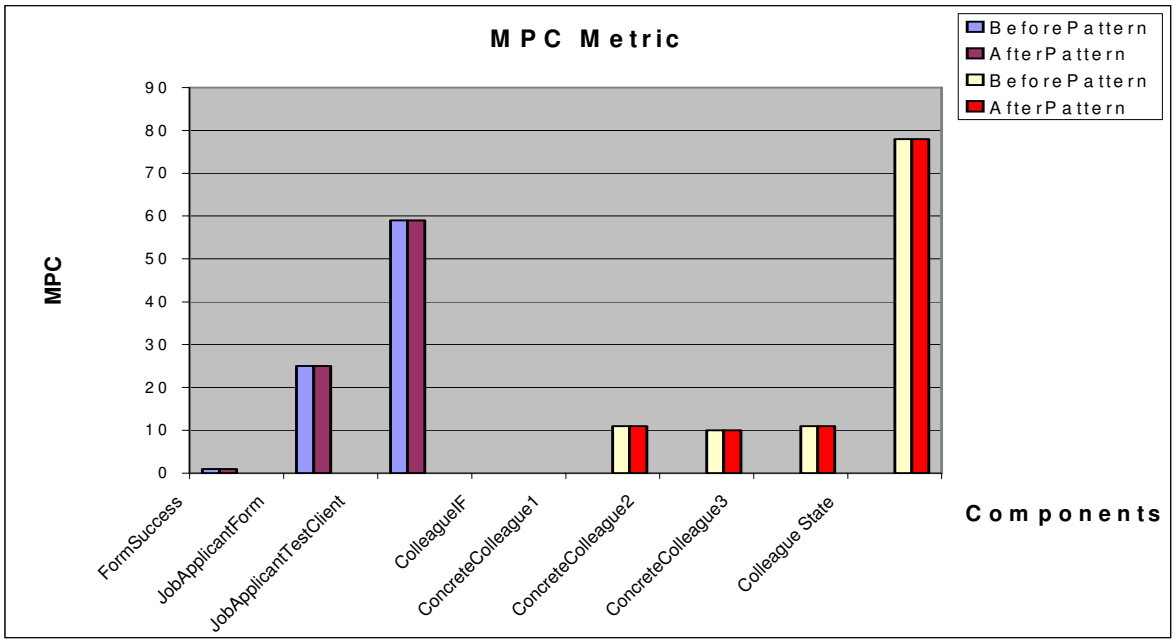


Figure 49 MPC for the case studies on Job Application and Colleague States

6.7 Size of change

The *size of change* $SC=[sc_{ij}]$ is defined as the ratio between the number of affected methods of the receiving component caused by the changes in the interface elements of the providing components and the total number of methods in the receiving component. For every component C_j, we associate the set M_j of the methods of component C_j, as shown in Figure 50. We determine the *effect coefficient* value μ_m^{ij} for every method *m* in component C_j, $j \neq i$. They take binary values:

- μ_m^{ij} =1, if the method *m* is affected by any interface element $v \in V_i$ provided by C_i
- μ_m^{ij} =0, otherwise.

The *size of change* sc_{ij} can be estimated:

$$sc_{ij} = \frac{1}{|M_j|} \sum_{m \in M_j} \mu_m^{ij} \qquad (4.6)$$

where $|M_j|$ is the cardinality of the methods set of component C_j.

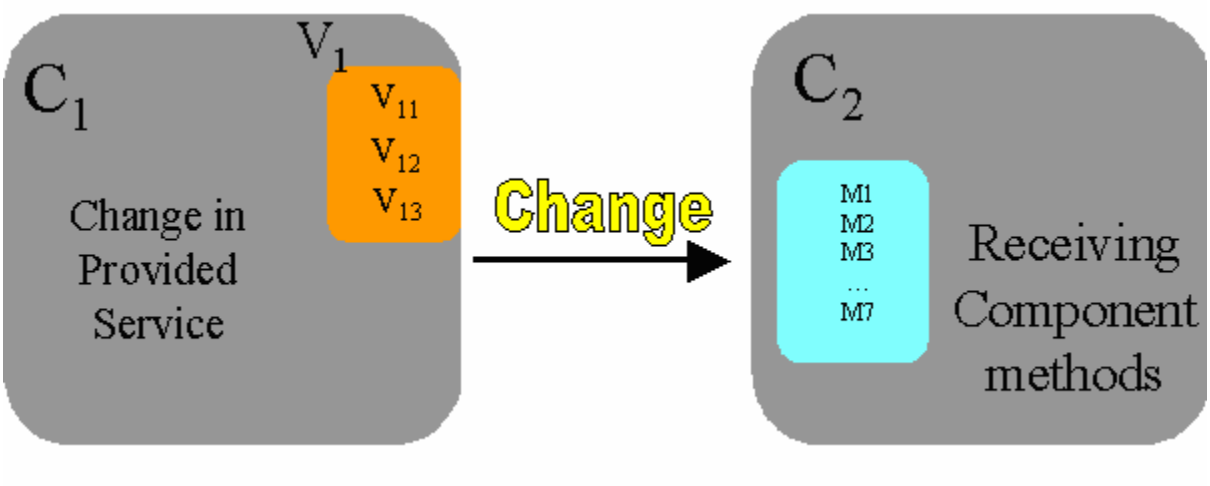


Figure 50 Size of change estimation

6.8 Change Propagation Probabilities and Size of Change for the Case Studies

In the next chapter we assess maintainability- based risk factors for the components of the system. Therefore, we need to estimate change propagation probabilities and size of change for the case studies under investigation. Using the software architecture artifacts of the pace maker, CM1 and the command and control system case studies, we estimate the change propagation probabilities and the size of change for each case study. Figure 51 and Figure 52 show the change propagation probabilities and the size of change of the pace maker. The change propagation probabilities and the size of change for the CM1 case study are shown in Figure 53 and Figure 54. While Figure 55 and Figure 56 give the change propagation probabilities and the size of change of the command and control case study.

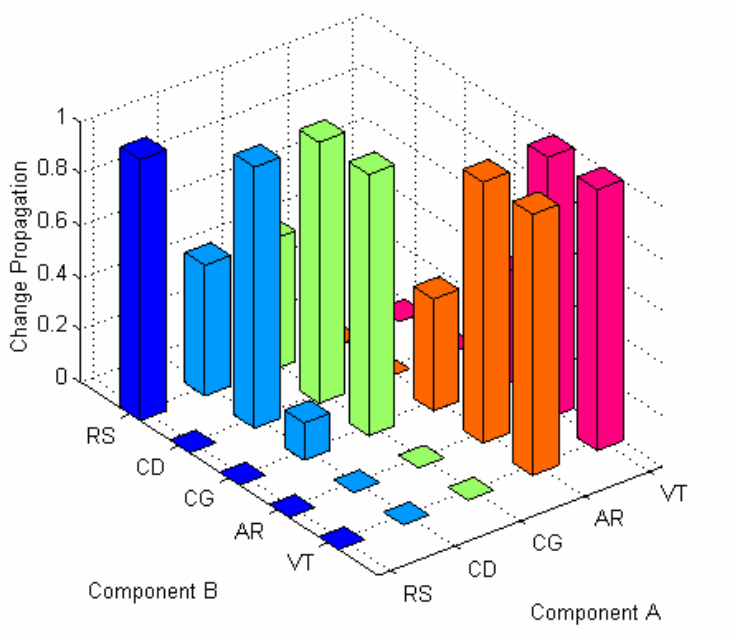


Figure 51 Change propagation probabilities for Pace Maker case study

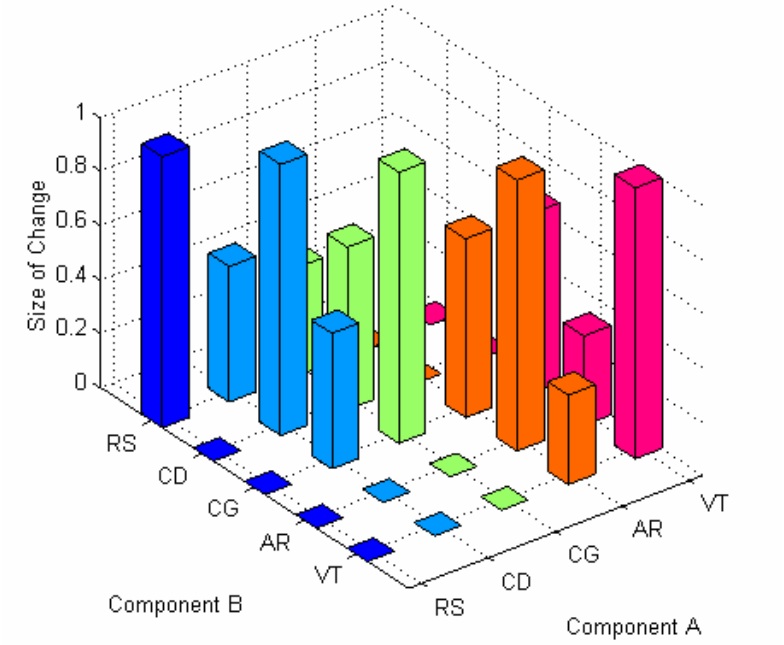


Figure 52 Size of change for Pace Maker case study

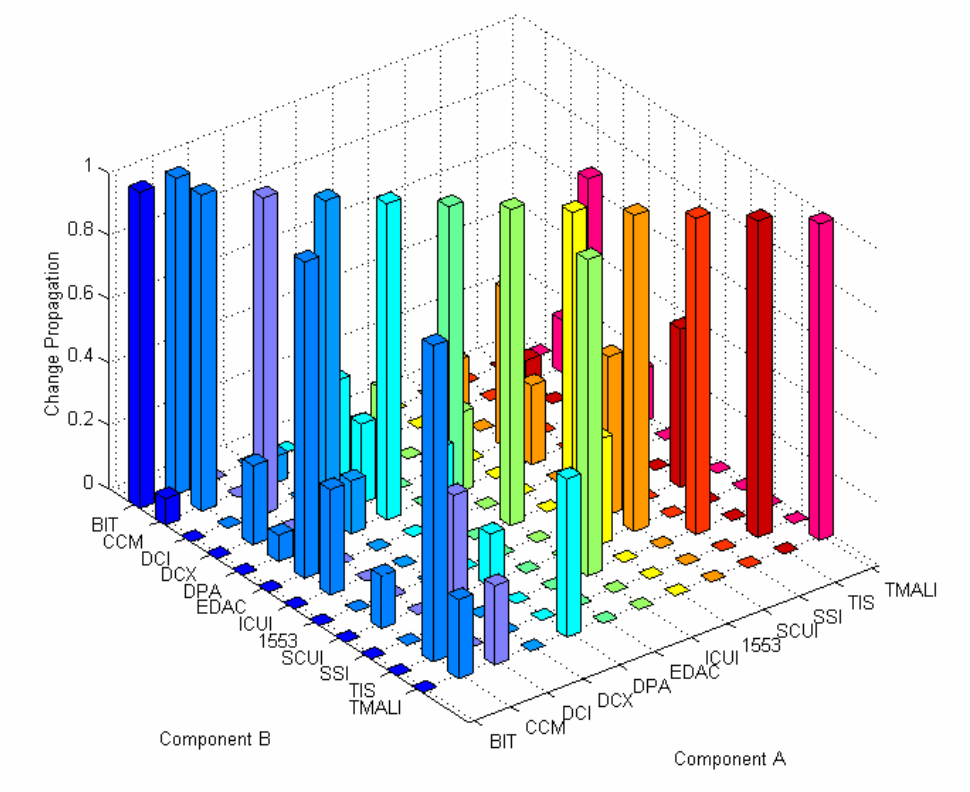


Figure 53 Change propagation probabilities for CM1 case study

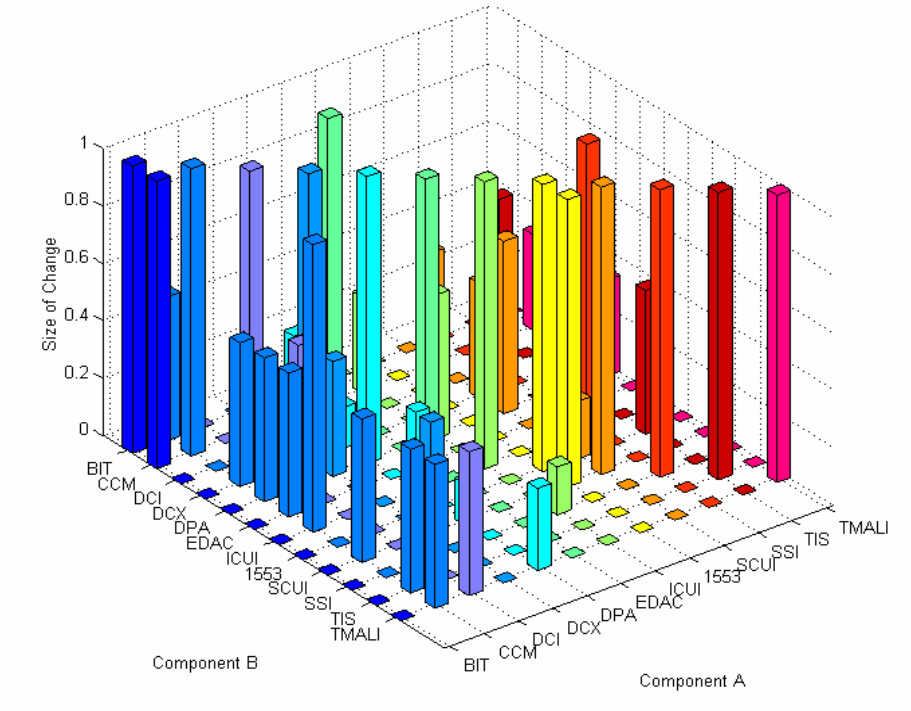


Figure 54 Size of change for CM1case study

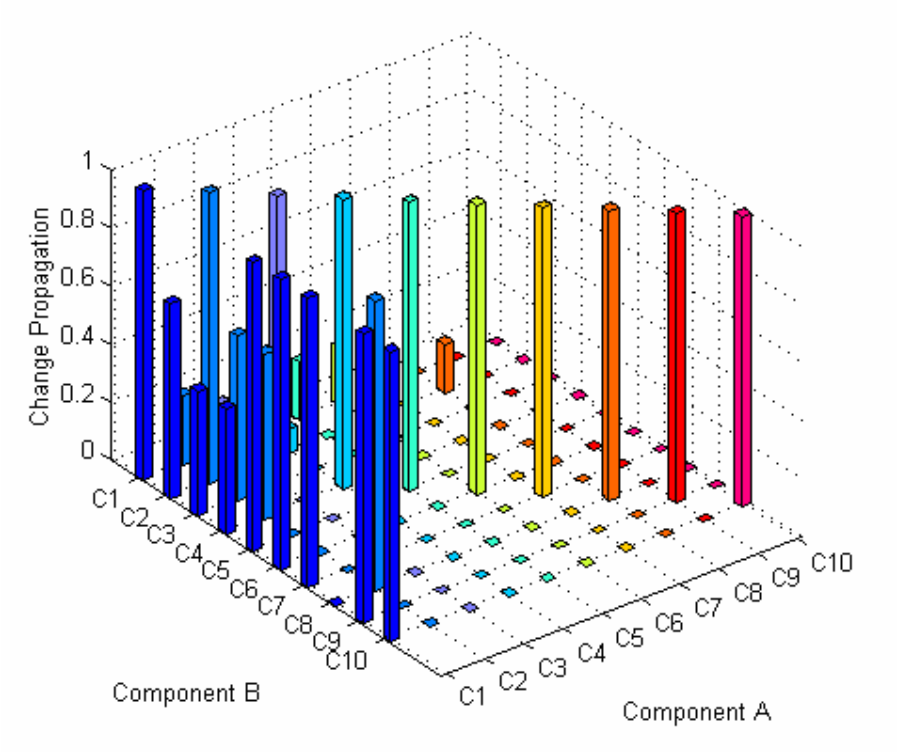


Figure 55 Change propagation probabilities for command and control case study

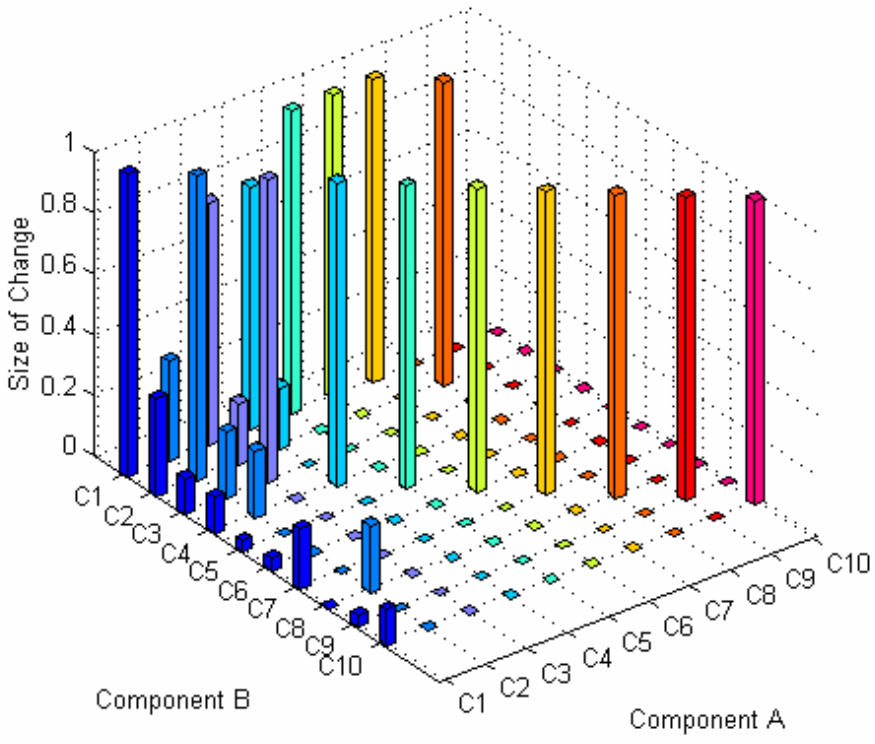


Figure 56 Size of change for command and control case study

6.9 Summary and Discussion

In this chapter, we proposed a set of simple, automatable formulas to quantify and assess change propagation probabilities through software architecture. For the purpose of validation of the analytical estimates, we conducted empirical experiment of *"mutation operators"*. We found a correlation between the two results for single-step change propagation probabilities of at least 0.85. To be certain that we didn't obtain these correlations by chance, we computed the t statistic to examine the statistical significance for these correlations. We got p-values < 0.05 which shows that the correlation between experimental and analytical error propagation matrices is statistically significant.

The proposed formulas are used to determine the impact of a change in a given component as a ripple, a wave, or an avalanche impact. The ripple change has minimal impact, a wave change has more impact but still controllable, and an avalanche change which is uncontrollable. We derived easy to compute upper bounds for n-step change propagation and cumulative change propagation, which can be used to measure the global impact of local changes.

In the following chapter, we will examine maintainability-based risk in perfective maintenance context. Refactoring the software using design patterns is a technique of conducting perfective maintenance. According to [Kerievsky 2004], there are practical situations where patterns improve the quality of the design. We conducted an experiment to assess to what extent the object-oriented metrics on one hand and the change propagation probability matrices on the other hand are good predictors of quality attributes. The experiment compared two candidate architectures: one that is based on design patterns and one that is design ad-hoc, using object-oriented metrics and change propagation matrices. Change propagation probabilities showed better ability to capture improvements in the design.

Also, we introduced the concept of size of change, which tries to capture the maintenance impact on a certain component given that other components coupled to it changes their interfaces. To check the validity of the proposed estimate for the size of change metric, we plan to conduct empirical experiment on several case studies noting that these kinds of empirical studies are effort and time consuming.

7 Maintainability-Based Risk Assessment

Software maintenance accounts for a large part of the life cycle software cost. Systems with good maintainability can be easily modified to fix faults or to adapt to changing environments. In this Chapter, we are concerned with maintainability-based risk that assesses how difficult it is to maintain the system in the future because of current maintenance tasks. We estimate maintainability-based risk of system components taking into consideration different types of maintenance: corrective, adaptive or perfective.

7.1 Maintainability-based Risk

In accordance with NASA-STD-8719 standard [NASA 1997], we define maintainability base risk is as a combination of two factors: the probability performing maintenance tasks and the impact of performing these tasks [Abdelmoez+ 2005B]. Accordingly, Maintainability-based Risk for a component is defined as:

Probability of changing the component Maintenance impact of changing the component.*

Maintainability-based risk assessment helps in managing software maintenance process. It can be used to identify the most risky parts of the system.

7.2 Estimation Methodology of Maintainability-based Risk

The proposed methodology for estimating maintainability-based risk depends on architectural artifacts such as system requirements and system design and their evolution through the life cycle of the system, as shown in Figure 57. First, we estimate initial change probabilities of the components according to the maintenance type and available data. Using the initial change probabilities of the components and change propagation probabilities between them, we get the unconditional probability of change of the components of the system. To get the impact of the maintenance tasks, we use the size of change between the components of the system. Finally, the maintainability-based component risk factor is the product of unconditional change probability and the maintenance impact. The detailed steps of the proposed methodology in the following subsections are presented using UML models [UML 2005].

7.2.1 Estimating Initial Change Probabilities

The maintenance effort can be due to corrective, perfective or adaptive maintenance. According to the type of effort considered, we estimate the initial change probabilities *ICP* of the system components. We use error reports, change reports, system requirement enhancements or requirements stability indexes to estimate *ICP* depending on the type of maintenance and the data available.

7.2.2 Estimating Change Propagation Probabilities

Change propagation probability $CP= [cp_{i/j}]$ is the conditional probability that a change originating in one component of the architecture requires changes to be made to other components. To account for the dependency among the components of the system, we multiply the initial change probabilities vector of the components by the conditional change propagation probabilities obtained from the system architecture.

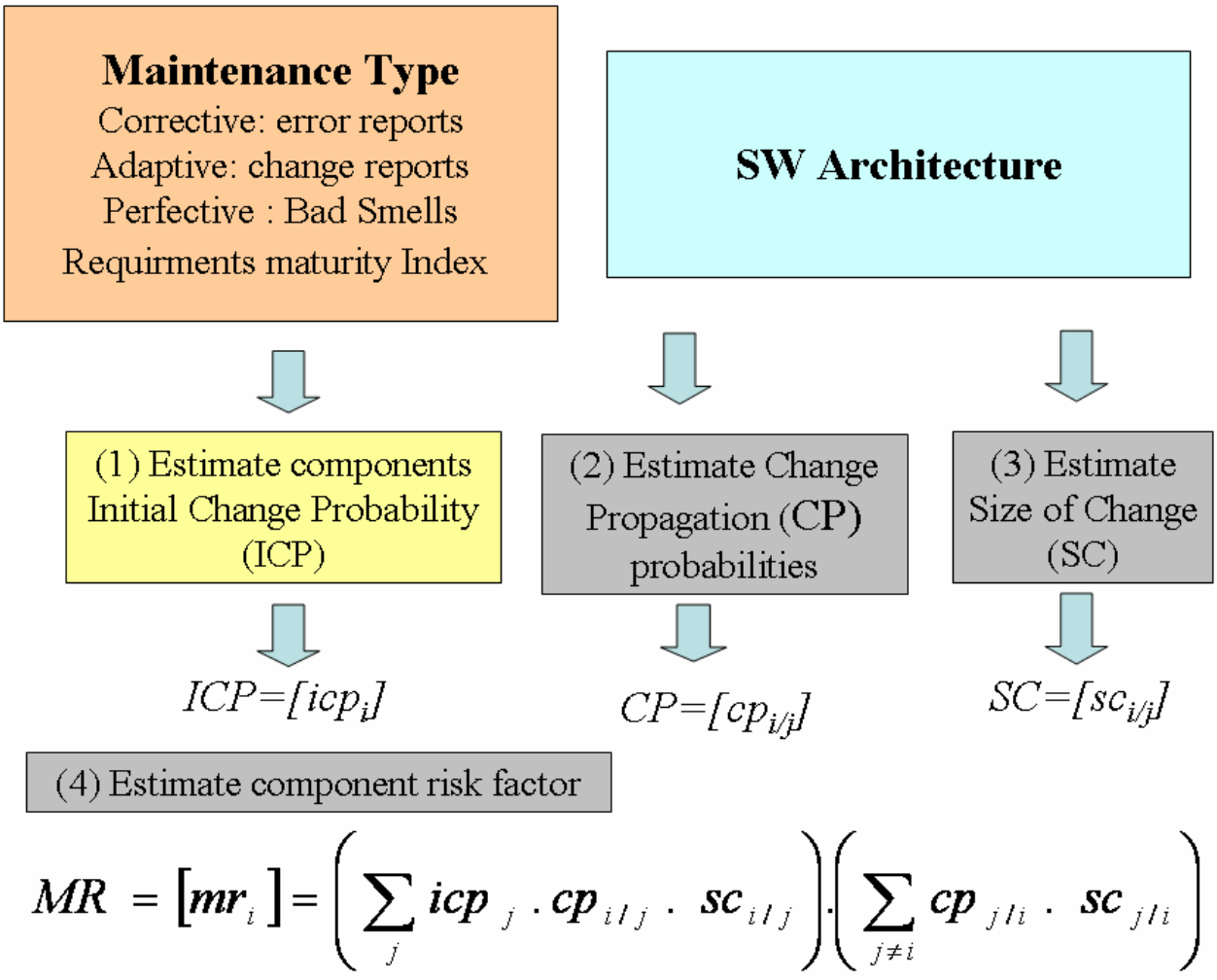


Figure 57 Maintainability-based risk estimation methodology

7.2.3 Estimating Size of Change

To get the impact of the maintenance task, we use the size of change between pairs of the system components. We define the size of change $SC=[sc_{i/j}]$ as the ratio between the number of affected methods

of the receiving component caused by the changes in the interface of the providing components and the total number of methods in the receiving component.

7.2.4 Estimating Components Maintainability-based Risk

For estimating the maintainability, we try to capture the maintenance change propagation shown in Figure 58 and Figure 59. If we consider change propagation through component C_i , the initial change in other components C_j are propagated to component C_i with change propagation probability $cp_{i/j}$ and size of change $sc_{i/j}$, as shown in Figure 58. Furthermore, the initial change in component C_i and the changes propagated from other components to component C_i propagate once again to other components C_j with change propagation probability $cp_{j/i}$ and size of change $sc_{j/i}$, as depicted in Figure 59. Thus, the maintainability-based risk MR is given by

$$MR = [mr_i] = \left(\sum_j icp_j \,.\, cp_{i/j} \,.\, sc_{i/j} \right) . \left(\sum_{j \neq i} cp_{j/i} \,.\, sc_{j/i} \right) \qquad (7.1)$$

Hence, the methodology provides the maintainer with an estimate of the maintainability risk of the components.

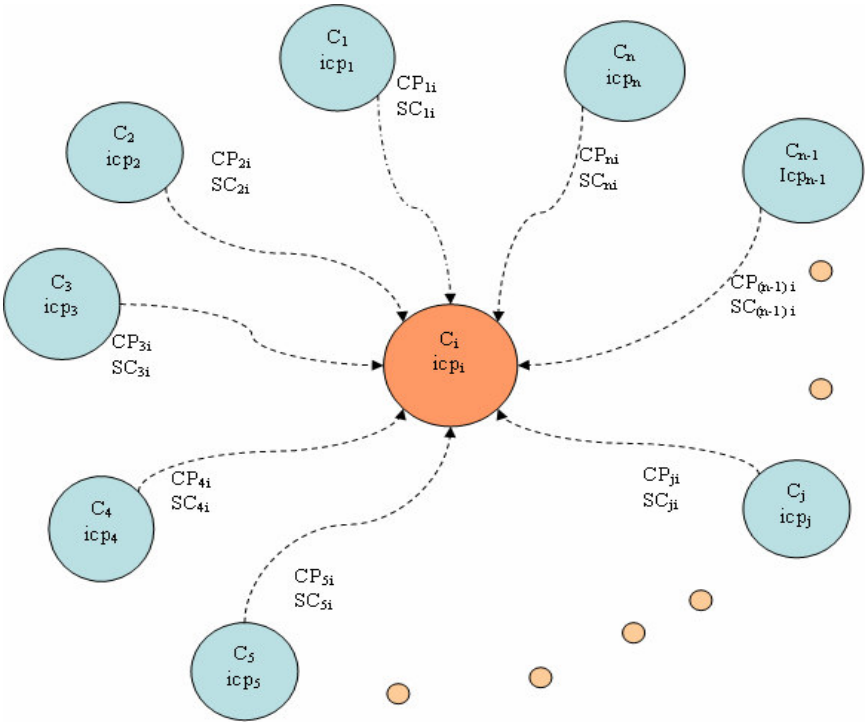


Figure 58 Incoming maintenance change propagation through component Ci

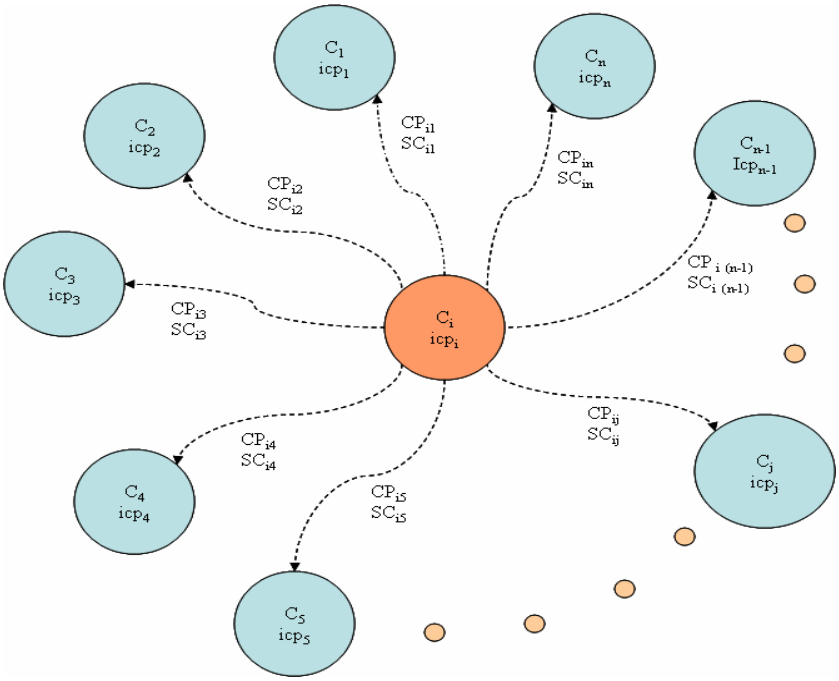


Figure 59 Outgoing maintenance change propagation through component Ci

7.3 Maintainability-Based Risk Assessment in Adaptive Maintenance Context

In this section, we limit our scope of maintenance effort to adaptive maintenance [Abdelmoez+ 2006B]. Thus, we alter the methodology for estimating the maintainability-based risk of software components as follows. Basically, we make use of adaptive maintenance reports of changes to estimate the initial change probabilities $ICP=[icp_i]$. First, we evaluate the rate of occurrence of changes in each component C_i of the system. Then, we estimate the initial probability of change for each component by normalizing the rate of occurrence for each component by the total number of change reports. Hence, the estimation methodology of maintainability-based risk is tailored for adaptive maintenance.

To take into consideration the dependency between the components of the system, we estimate the conditional change propagation probabilities matrix *CP* and the size of change *SC* from the system architecture. Finally, the maintainability-based risk of a component C_i due to adaptive maintenance changes mr_i is estimated using equation (7.1). We propose to use the maintainability-based risk of the system components to order the adaptive maintenance tasks for a certain project.

7.3.1 CM1 Maintainability-Based Risk in Adaptive Maintenance Context

The maintenance data of the CM1 case study contain 31 change reports. (See the details of the case study in Appendix I.C). We want to prioritize the tasks of the adaptive maintenance effort. First, we

calculate the frequency of requested change occurrences in the components of the system. Second, we estimate the initial change probability ICP of the components of CM1 by normalizing the frequency of change occurrences by the total number of change reports. The estimated initial change probabilities ICP for CM1 components are shown in Figure 60.

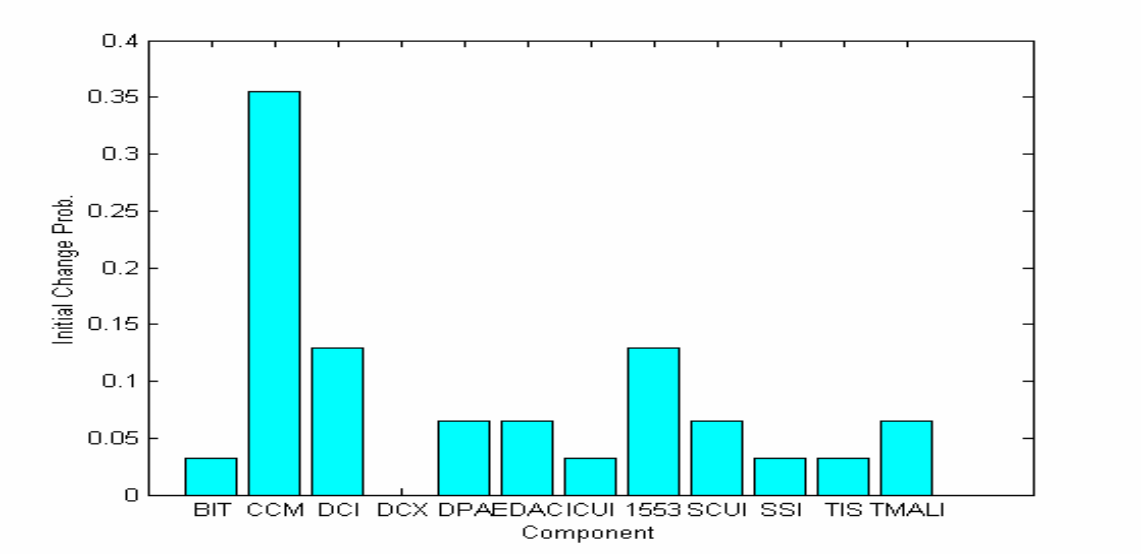


Figure 60 Initial change probabilities for CM1 components

Then using the software architecture artifacts of CM1, we estimate the change propagation probabilities and size of change, as shown in Figure 53 and Figure 54. Using equation (7.1), the maintainability-based component risk factor for each CM1 component is estimated. The results are shown in Figure 61.

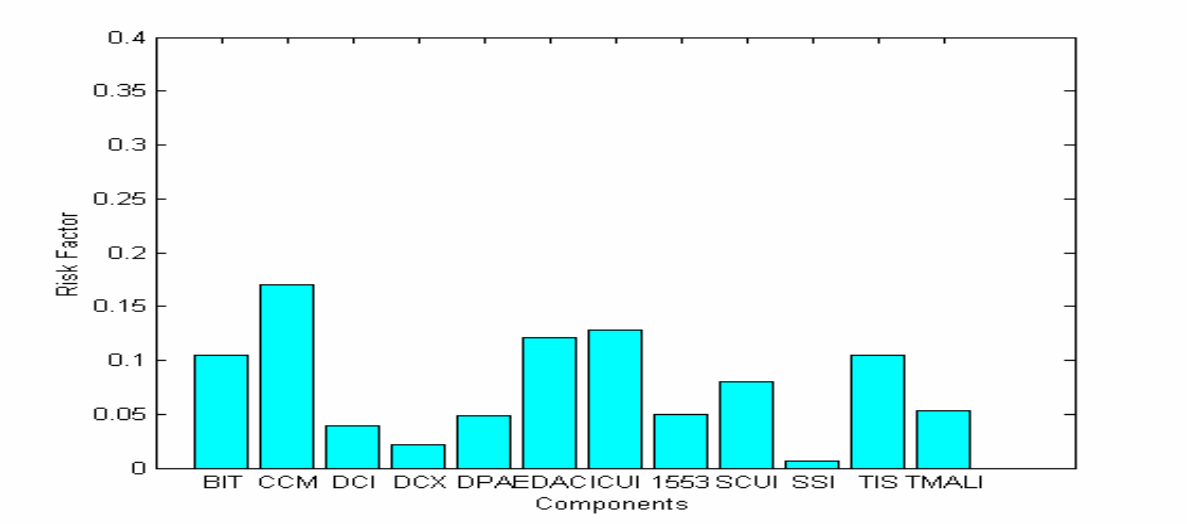


Figure 61 Maintainability-based risk for CM1 components in adaptive maintenance context

The most risky component with respect to adaptive maintenance is CCM. This is a result of CMM having the highest initial change probability. Moreover, CCM is coupled to most of the components, so it is likely to be affected by the changes introduced in these components (CP values are high). Furthermore, CMM has a high maintenance impact on the rest of CM1 components (Σsc_{ij} is large). As it is coupled to other components in the system, the change is likely to propagate further.

On the other hand, even though component 1553 has a relatively high initial change probability value, but it is coupled to a limited number of components in CM1 (CP values are low). Moreover, it has a limited maintenance impact (Σsc_{ij} is small) and it is less risky in terms of maintainability. On the contrary, component EDAC has a relatively low initial change probability value. But due to change propagation (CP values are relatively high) and maintenance impact (Σsc_{ij} is not small), it is more risky in terms of maintainability

7.4 Maintainability-Based Risk due to Requirements Changes

In this subsection, we are addressing changes in the system requirements [Abdelmoez+ 2006A]. We propose an estimation methodology for the maintainability-based risk using UML models, which are becoming a de facto standard for modeling software systems. The detailed steps of the maintainabilitiy-based risk methodology are adapted to fit the adaptive maintenance context. First, we estimate the requirement maturity by analyzing their evolution across the versions of the system. A software system is developed according to a set of requirements

$$RQ = \{rq_1, rq_2, \ldots\ldots rq_p\} \tag{7.2}$$

where rq_i is a functional requirement. In UML, requirements are mapped into a set of use cases:

$$RQ(\text{UML}) = \{uc_1, uc_2, \ldots\ldots uc_p\} \tag{7.3}$$

Use cases describe the functional behavior of the system. Each use case is realized through one or more sequence diagrams. Sequence diagrams describe the interactions among the components to fulfill certain requirement. Since it is not possible to account for all possible maintenance tasks, we only consider a maintenance profile *MP* [Bosch+ 2001] consisting of likely change scenarios

$$MP = \{cs_1, cs_2, \ldots\ldots cs_s\} \tag{7.4}$$

A change scenario is defined by a set of requirement changes

$$cs_i = \{rq_{1c}, rq_{2c}, \ldots\ldots rq_{tc}\} \tag{7.5}$$

where rq_{ic} is an addition, deletion or modification of use case $uc_{i.}$

The IEEE 982 standard [IEEE Std 982.1] suggests *Software Maturity Index* to quantify properties of requirements evolution. In [Anderson+ 2002], The Software Maturity Index is adapted to *Requirements Maturity Index* (*RMI*) to measure the requirements stability. We adapt the metric to *Use Case Maturity Index* (*UCMI*) and use function points as a size measure for the use cases [Cantone+ 2004]. Thus, the *UCMI* of the use case uc_i is given by

$$UCMI = \frac{U_T - U_C}{U_T} \tag{7.6}$$

where U_T is the function point size of the use case uc_i in the current release; U_C is the function point size of the change in the use case uc_i in the current release from the previous due to requirement change rq_{ic} of change scenario cs_m.

In order to get the probability of change due to a maintenance task, we use the sequence diagrams to get the set of components that contribute to each use case. Then, we can map the use case stability into components stability, which reflects on the likelihood of making changes to the components due to changes in the requirements. Consequently, we estimate Initial Change Probabilities ICP of the system components. For components that are part of multiple scenarios, we consider the maximum ICP as it accounts for the worst-case scenario.

To account for the dependency among the components of the system, we multiply the initial change probabilities vector ICP of the components by the conditional change propagation probabilities matrix CP obtained from the system architecture. To get the impact of the maintenance task, we estimate the size of change SC=[sc_{ij}] between pairs of the components of the system based on the architecture artifacts. Finally, the components maintainability-based risk MR = [mr_i] can be estimated using equation (7.1), where mr_i is maintainability-based risk of a component C_i due to requirement changes. Hence, the methodology provides the maintainer with an estimate of the maintainability risk of the components for different change scenarios of the maintenance profile.

7.4.1 CM1 Maintainability-Based Risk due to Requirements Changes

We illustrate our risk assessment methodologies on CM1 case study from the Metrics Data Program [NASA MDP]. The details of the case study are in Appendix I.C. From the use case model in Figure 96, we identify the set of functional requirement RQ as:

$$RQ(\text{CM1}) = \{\text{Transfer, RecvCmd, ChBound, CalcOrbitDrift, HeartBeat,}$$
$$\text{HouseKeeping, TimeSync}\} \tag{7.7}$$

We estimate the requirement maturity by analyzing their evolution across the versions of the system. As, it is not possible to account for all possible maintenance tasks; we only consider a maintenance profile *MP*. To make it easier to follow the steps of the methodology, we consider only a maintenance profile that has only one change scenario.

$$MP = \{cs_1\} \tag{7.8}$$

The change scenario that we picked is adding a new transfer sequence, shown in Figure 98, to the *Transfer* use case:

$$cs_1 = \{ Transfer_c \} \tag{7.9}$$

We measure the function point size of the use case *Transfer* in the current release; and the function point size of the changes in the use case *Transfer* in the current release from the previous due to change $Transfer_c$ of change scenario cs_1. We follow the rules presented in [Cantone+ 2004] to estimate the function point size of the *Transfer* use case and the change in it. Then, we estimate the use case maturity index, according to equation (7.6). We find *UCMI(Transfer)=0.702.*

We map the use case maturity index into components stability using the sequence diagram $Transfer_c$. We determine how each component stability been affected according to its amount of contribution in the added sequence diagram $Transfer_c$. Then, we estimate initial change probabilities of the components. The results are shown in Figure 62.

We estimate the initial change probabilities of the components and change propagation probabilities between them the components of CM1. Then, we estimate the size of change between the components to account for the maintenance impact. Using equation (7.1), The maintainability-based component risk factor for CM1 is estimated. The results are shown in Figure 63. The most risky components are DPA and DCX as they have the highest initial change probability value when considering the change scenario cs_1.

It worth noting that component as CCM has a significant level of risk factor even though it is not in the set of components of the initial change. This is due to the fact that CCM is coupled to all of the components of the initial change set, so it is likely to be affected by the changes introduced in these components. Furthermore, CMM has a high maintenance impact on the rest of CM1 components. As it is coupled to other components in the system other than the components of the initial change set, the change is likely to propagate further.

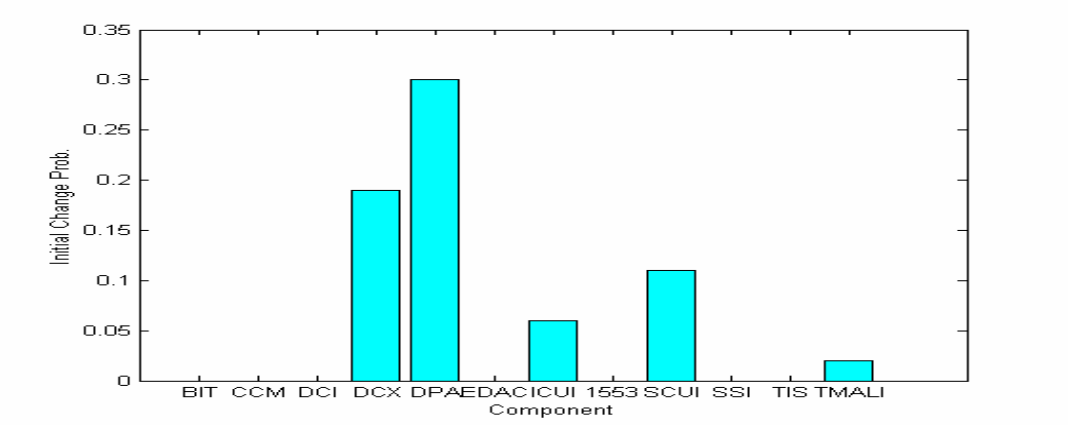


Figure 62 Initial change probabilities resulted from $Transfer_c$ for CM1 components

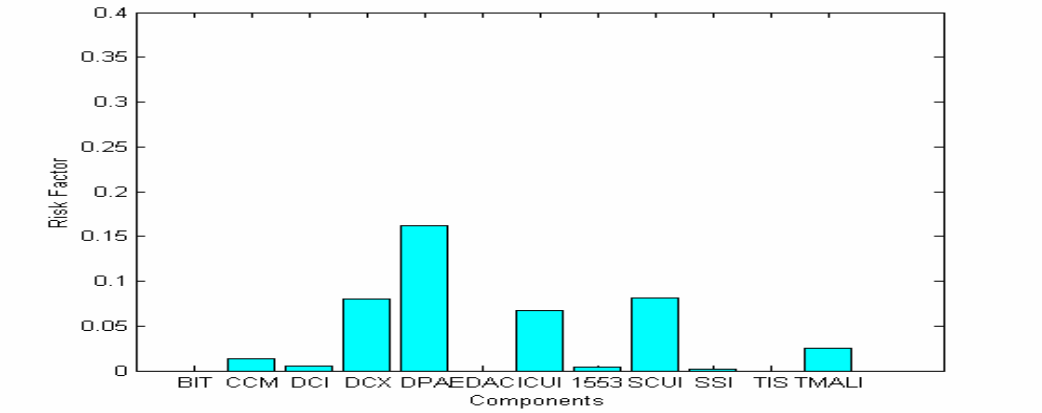


Figure 63 Components maintainability- based risk resulted from $Transfer_c$ for CM1 components

7.5 Maintainability-Based Risk Assessment in Corrective Maintenance Context

In this section, we limit our consideration of maintenance effort to corrective maintenance [Abdelmoez+ 2006C]. Therefore, we use error reports of errors that have not been yet fixed. To estimate the initial change probabilities $ICP=[icp_i]$, we first evaluate the frequency of occurrence of errors in each component C_i of the system. Then, we estimate the initial probability of change for each component by normalizing the frequency of occurrence for each component by the total number of error reports. Hence, the estimation methodology of maintainability-based risk is adapted for corrective maintenance.

To order the corrective maintenance tasks for a certain project according to the importance of each task, we propose using the maintainability-based risk of the components that need to be fixed. Also, we propose to consider the severity-level of failures that may be manifested from the errors in these components. For maintenance tasks of components with critical or catastrophic severity-levels, the

maintainability-based risk should not be of concern because of the consequences of such potential failures on the system. Such tasks should be of high priority in the maintenance plan. On the other hand for maintenance tasks of components that have severity-levels of minor or major, we should examine the components maintainability-based risk. So, maintenance tasks of low severity-level and high maintainability-based risk should be avoided or delayed in the maintenance plan. But if the priority is to have a system with low maintainability risk, we start with the components with the high level of maintainability-based risk and fix them into more maintainable components. Thus, we can prioritize the maintenance tasks accordingly.

7.5.1 CM1 Maintainability-Based Risk Results

In the following, we present the results of the severity analysis and the maintainability-based risk for the CM1 case study. Then, we discuss the results and how to prioritize corrective maintenance tasks based on both of severity level and maintainability-based risk. The CM1 case study has 98 error reports of components bugs. Assuming that these errors have not been yet fixed, we want to prioritize the tasks of the corrective maintenance effort. First, we calculate the frequency of errors occurrences in the components of the system, as shown in Table 12. Second, we estimate the initial change probability ICP of the components of CM1 by normalizing the frequency of error occurrences by the total number of error reports. The estimated initial change probabilities ICP for CM1 components are shown in Figure 64. Then using the software architecture artifacts of CM1, we estimate the change propagation probabilities and the size of change, as shown in Figure 53 and Figure 54.

Table 12 Components error reports of the CM1 case study

	Components											
	BIT	CCM	DCI	DCX	DPA	EDAC	ICUI	1553	SCUI	SSI	TIS	TMALI
Errors	0	15	11	0	5	5	6	4	13	14	4	21

Using equation (7.1), the maintainability-based component risk factor for each CM1 component is estimated. The assignment of component severity level of each component is based on the hazard analysis conducted by domain experts knowledgeable about the case study (See Section 4.5.2). The results are shown in Figure 65.

For planning corrective maintenance of this system, we should think about components maintainability-based risk. Also, we should take into consideration the severity level of potential failures

that could be caused by errors in components needed to be fixed. CMM, DCI, 1553 and SSI components with catastrophic severity-levels and DPA, EDAC and TMALI components with critical severity-levels, they should be fixed regardless of their corresponding maintainability-based risk because of the consequences of such potential failures on the system. On the other hand for maintenance tasks of the rest of the components having low severity-levels, we should examine the components maintainability-based risk.

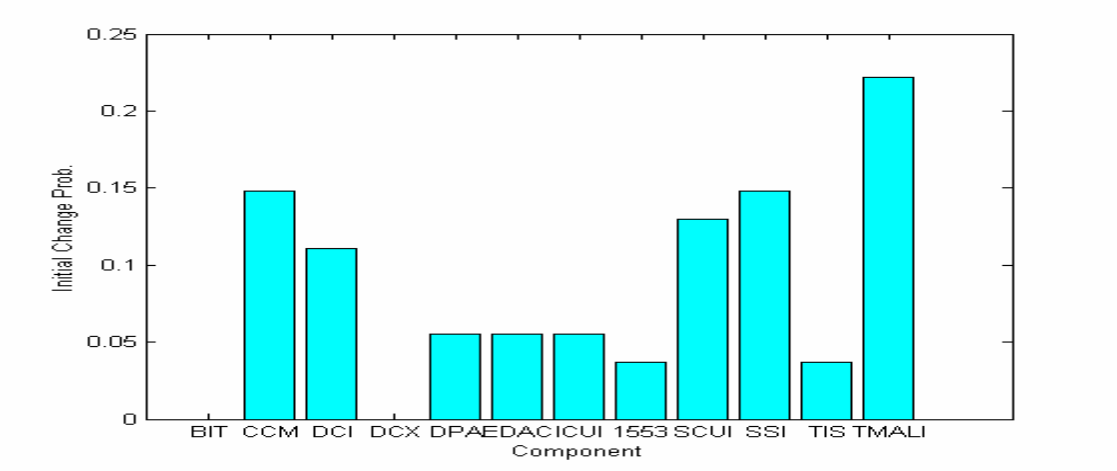
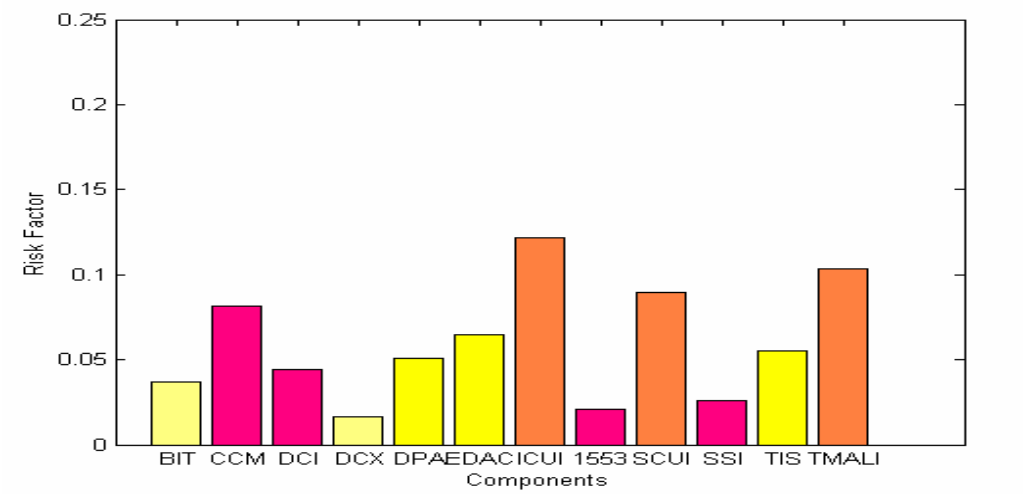


Figure 64 Initial change probabilities for components of CM1 case study



	Components											
	BIT	**CCM**	**DCI**	**DCX**	**DPA**	**EDAC**	**ICUI**	**1553**	**SCUI**	**SSI**	**TIS**	**TMALI**
Severity Level	Minor	Cat.	Cat.	Minor	Major	Major	Critical	Cat.	Critical	Cat	Major	Critical

Figure 65 Maintainability- based risk and severity levels for CM1 components

7.5.2 Pace Maker Maintainability-Based Risk Results

As we don't have error reports for the pace maker, we estimate the initial change probability ICP of the components by normalizing the components cyclomatic complexity by the total sum. The estimated initial change probabilities ICP for pace maker components are shown in Figure 66. Then using the software architecture artifacts of pacemaker, we estimate the change propagation probabilities and the size of change, as shown in Figure 51 and Figure 52.

Using equation (7.1), the maintainability-based component risk factor for each pace maker component is estimated. The assignment of component severity level of each component is based on the hazard analysis conducted by domain experts knowledgeable about the case study (See Section 4.5.1). The results are shown in Figure 67.

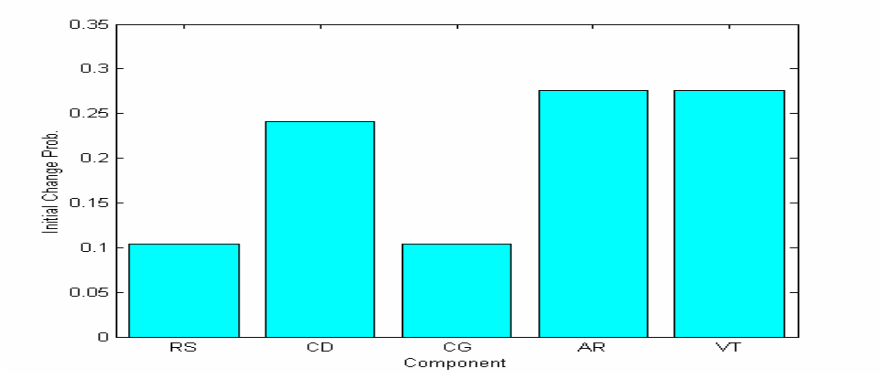
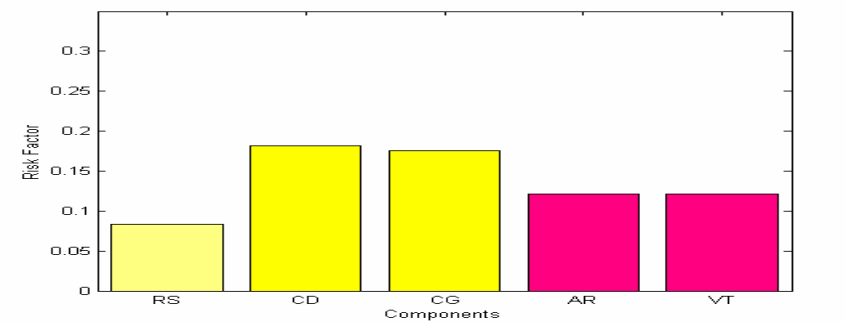


Figure 66 Initial change probabilities for components of PM case study



	Components				
	RS	**CD**	**CG**	**AR**	**VT**
Severity Level	Minor	Minor.	Major.	Catastrophic	Catastrophic

Figure 67 Maintainability- based risk and severity levels for pace maker components

For prioritizing maintenance tasks for this pace maker in corrective maintenance context, we should consider the followin:.

- AR and VT components with catastrophic severity-levels, they should be fixed regardless of their corresponding maintainability-based risk because of the consequences of such potential failures on the system.
- On the other hand for maintenance tasks of the RS and CD components having minor severity-levels and CG component with major severity-level, we should examine the components maintainability-based risk.

7.5.3 Command and Control System Maintainability-Based Risk Results

For the command and control system case study, we also don't have error reports. So, we estimate the initial change probability ICP of the components by normalizing the cyclomatic complexity of the system components by their total sum. The estimated initial change probabilities ICP for command and control system components are shown in Figure 68. Using the software architecture artifacts of command and control system, we estimate the change propagation probabilities and the size of change, as shown in Figure 55 and Figure 56.

Using equation (7.1), the maintainability-based component risk factor for each command and control system component is estimated. The assignment of component severity level of each component is based on the hazard analysis conducted by domain experts knowledgeable about the case study (See Section 4.5.3). The results are shown in Figure 69. We can recognize that other than components C1 and C2, the maintainability-based risk factors are quite small. That is because these components haves mall values of change propagation probabilities. Furthermore, they are not highly coupled with each other. They are mainly coupled to component C1 or/and C2. Thus, all change propagation of maintenance tasks will affect components C1 and C2.

For prioritizing maintenance tasks for this system in corrective maintenance context, we should consider the following. First, C1 and C2 components have catastrophic severity-levels and C3, C4 and C7 component have critical severity-level. They should be fixed regardless of their corresponding maintainability-based risk because of the consequences of such potential failures on the system. On the other hand for maintenance tasks of the rest of the components having low severity-levels, we should examine the components maintainability-based risk.

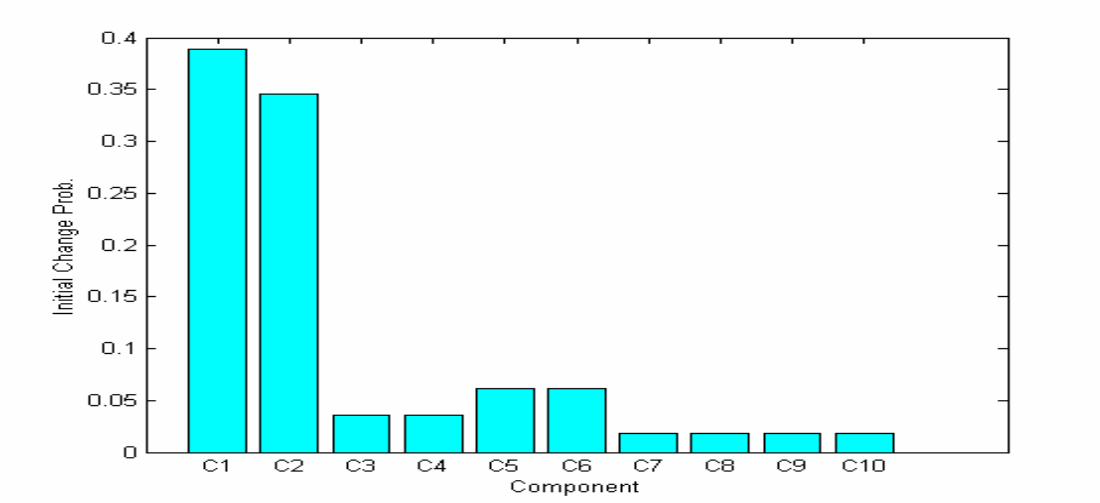
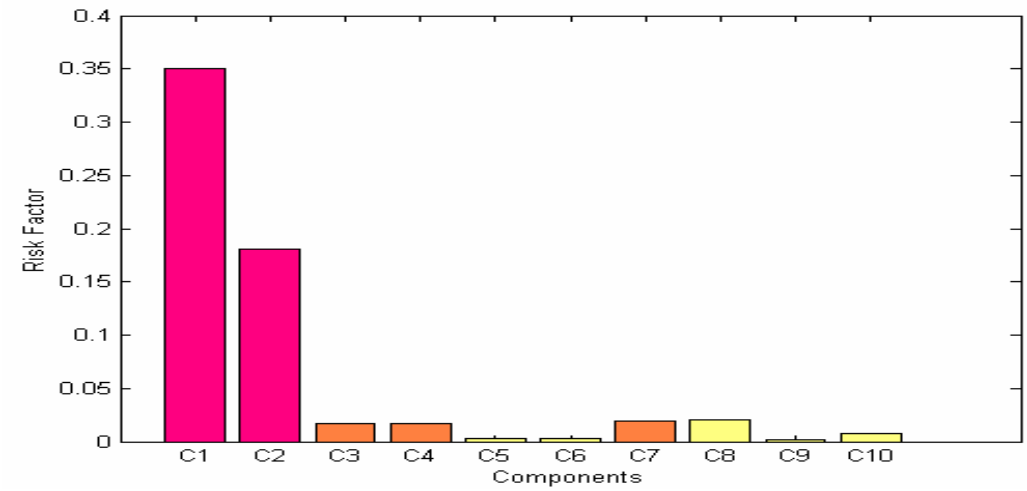


Figure 68 Initial change probabilities for components of command and control case study



	Components									
	C1	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
Severity Level	Cat.	Cat.	Critical	Critical	Major	Major	Critical	Minor	Major	Minor

Figure 69 Maintainability- based risk and severity levels for command and control components

7.6 Maintainability Based Risk in Perfective Maintenance Context

In this subsection, we focus on perfective maintenance and refactoring activities in particular. Refactoring is defined as a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviors. Refactoring to improve the design of the system requires knowing which parts of the system need to be improved. In [Fowler+ 1999],

Fowler and Beck presented a list of bad smells that help to identify where refactoring is needed. Examples of bad smells include large class, lazy class, data class and switch statements. Considering the architectural level, not all of the smells can be identified.

One way to refactor the software is to use design patterns. According to [Kerievsky 2004], there are practice situations where patterns help to improve the quality of the design. We improve maintainability by:

- reducing or removing duplication,
- simplifying what is complicated and
- making the design better at communicating its intent.

We use bad smells of the architecture to estimate components' maintainability-based risk. In particular, we consider two smells: divergent change and shotgun surgery.

Divergent change is when one component is commonly changed in different ways for different reasons [Fowler+ 1999]. For example, we have to modify the same component whenever we change the database or add a new calculation formula. To estimate divergent change smell using the CP matrix, we examine if we have high values in a column corresponding to a component C_i. Such a component is likely to undergo frequent changes in the maintenance phase due to changes in other components.

Shotgun surgery is when every time a change is made to a component; lots of little changes need to be made to a lot of different components [Fowler+ 1999]. For example, whenever we change a database we must change several components. To estimate shotgun smell using the CP matrix, we examine if we have high values in a row corresponding to a component C_i. Changes to such a component need to be avoided because they propagate throughout the system.

We use equation (7.1) to estimate components' maintainability risk of the original system and the refactored system after applying design pattern. We want to show that the maintainability based risk estimated based on the change propagation probabilities can capture improvements in the maintainability of the system components as result of introducing these design pattern. We use Strategy pattern and MVC pattern in our case studies.

The first case study is a simple application where an employer is seeking applications for the various jobs available. There are two versions; one version is a simple switch case whereas the other version is implemented using the strategy pattern. (The details of the case study are in Appendix I.D.3). We restrict the analysis to the components that exist before and after the application of the pattern. Figure 70 shows Components maintainability-based risk for the case study before and after applying the pattern. We see

that there are improvements in Components maintainability-based risk. The addition of the strategy pattern results in decoupling the FormSuccess and JobApplicantForm components. That causes the maintainability-based risk of these components to improve. These improvements in these components are associated with the cost of adding new components to the system.

The second case study is an open source calendar and task tracking software written in Java [Borg]. (The details of the case study are in Appendix I.D.4). Figure 71 shows components maintainability-based risk for the case study before and after implementing the controller of the MVC pattern. We can identify that errmsg is the most risky component. This component is responsible of showing an error message whenever an exception occurs. The risk factor of this component didn't change before and after adding the controller class of the MVC pattern because this modification does not address the errmsg component. There are improvements in some components' maintainability-based risk. On the other hand, there is deterioration in others. We restrict the analysis to the components that existed before adding the controller to the MVC pattern. The biggest improvement in maintainability risk factor is in borg component. This is a result of adding the controller class, as it causes the coupling of the borg component to decrease because it is redirected to the added controller class. Also, this modification causes other components to be more coupled like taskmodel and taskgui because they need to interact with the added controller class. Thus, it increases the maintainability risk factor of these components.

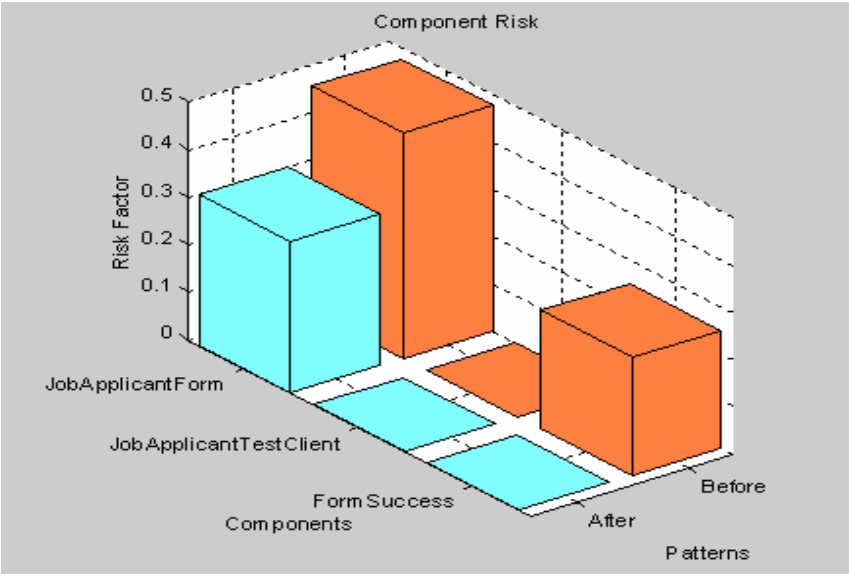


Figure 70 Components maintainability-based risk for job application case study.

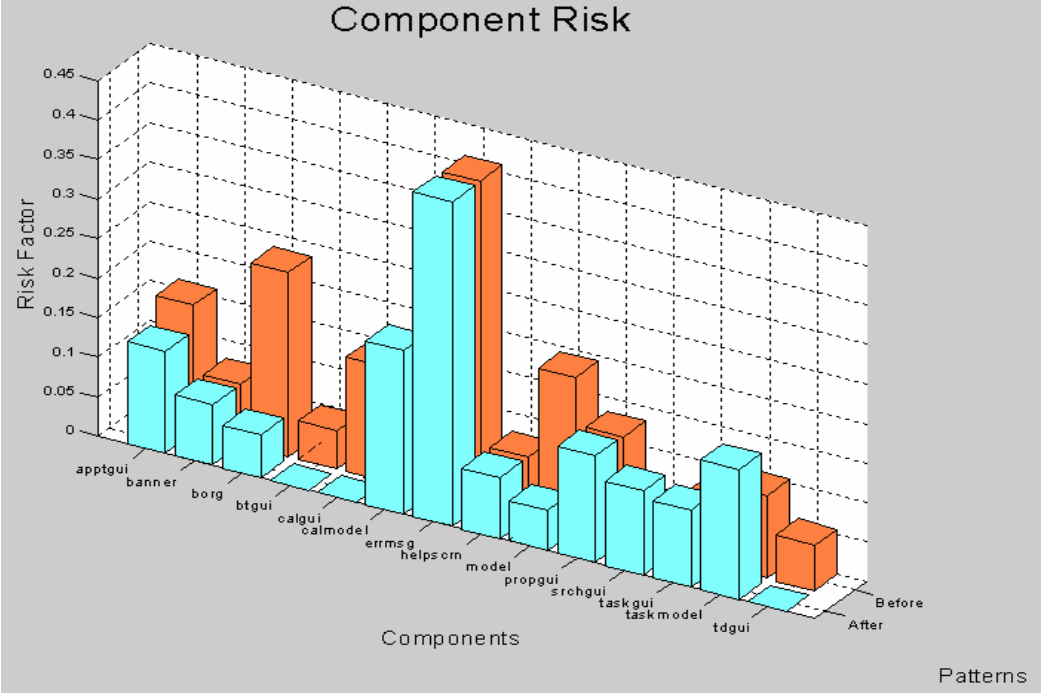


Figure 71 Components maintainability-based risk for the case study.

7.7 Worst Case Maintainability-Based Risk Estimate

It is helpful to conduct worst case analysis, for maintainability-based risk this corresponds to the case when the initial change probability equals to one for all the components of the system. This is the case when the system is totally unstable. Thus, we are certain that there will be maintenance changes to all of the components of the system. Figure 72, Figure 73 and Figure 74 show the worst case components maintainability-based risk for the pace maker, command and control system and CM1 case studies. Using the software architecture artifacts of the case studies, we estimate the change propagation probabilities and the size of change Then we use equation (7.1) for estimating the worst case components maintainability-based risk by substituting for ICP=[icp_i] with ones.

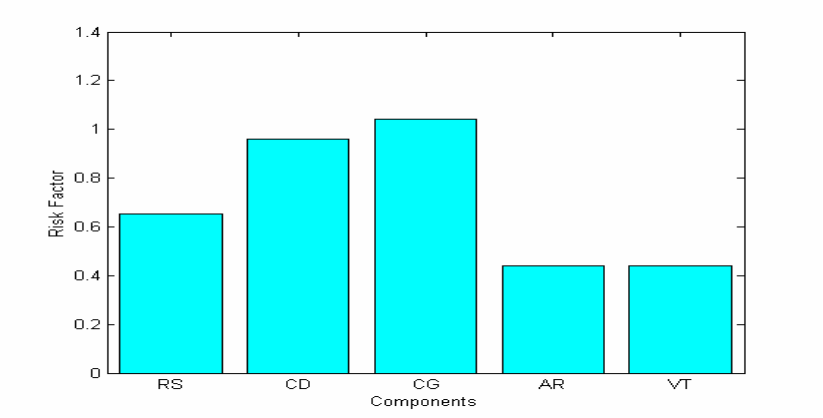


Figure 72 Worst-case Maintainability- based risk estimate for PM case study

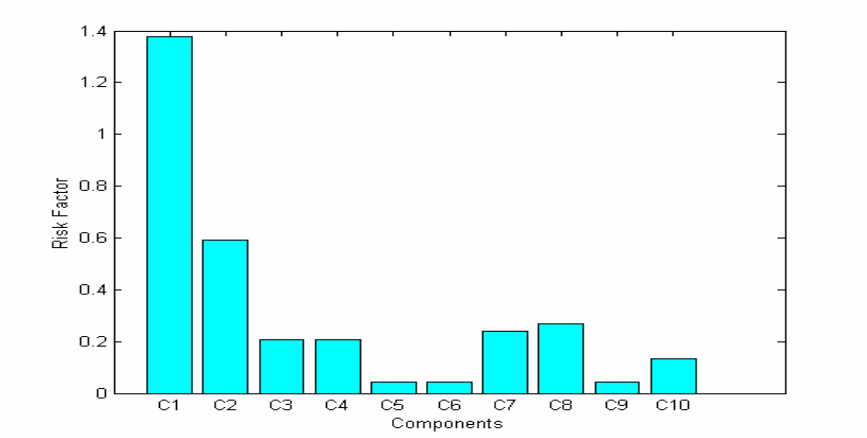


Figure 73 Worst-case Maintainability- based risk estimate for command and control case study

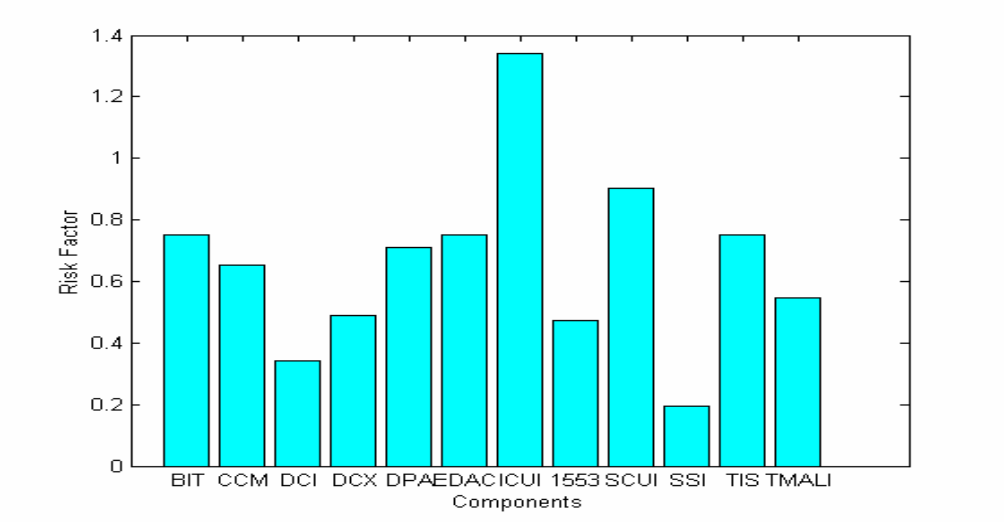


Figure 74 Worst-case Maintainability- based risk estimate for CM1 case study

7.8 Using Non-Homogeneous Poisson Process to Estimate Maintainability-Based Risk

Unlike the previous maintainability-based risk models, *Non Homogeneous Poisson Process NHPP* provide us with an estimate which is a function of time when considering adaptive and perfective maintenance (See section 3.3.3). The *NHPP* model captures the nature of maintenance request arrivals [Tan+ 2005]. Furthermore, the estimation procedure is more flexible as it relies on a statistical model to estimate initial change probability ICP at different points of time. As the system gains more stability through the development or the maintenance effort, we are able to acquire better estimates for the parameters of the *NHPP* statistical model. Thus, we can have a better predication of the maintainability level of the system under consideration

We simulate the random arrival rate of the maintenance request rates based on the model proposed by [Tan+ 2005]. We use the arrival rates to get an estimate of the initial change probability as a function of time according to the assumed adaptive maintenance for the system components and the perfective maintenance of the system features (i.e. modifying use cases of the model in context of perfective manitnenance). We estimate the initial change probabilities of the components of the CM1 case study. We consider adaptive maintenance and perfective maintenance modeled by a maintenance profile that has three change scenarios.

$$MP = \{cs_1 , cs_2 , cs_3 \} \tag{7.10}$$

We assume that these change scenarios consist of cs_1 for modifying the initial system features, cs_2 for modifying the transfer sequence $Transfer_c$, shown in Figure 98, and cs_3 for modifying *HeartBeat* shown in Figure 99:

$$cs_2 = \{ HeartBeat\}, cs_3 = \{ Transfer_c \} \tag{7.11}$$

These simulation settings are shown in Figure 75. We assume that change request for each change scenario is distributed among the contributing components in a uniform fashion. For example, the change scenario cs_2 for the *HeartBeat* use case have the following contributing components CCM, ICUI and SSI. Thus in the simulation setting for the maintenance request that is generated by the inhomogeneous Poisson process for the change scenario cs_2 are distributed uniformly among these components. For each component at any instance of the simulation time, we accumulate the generated request rate from all the sources of maintenance (adaptive maintenance and perfective maintenance for each feature introduced into the system).

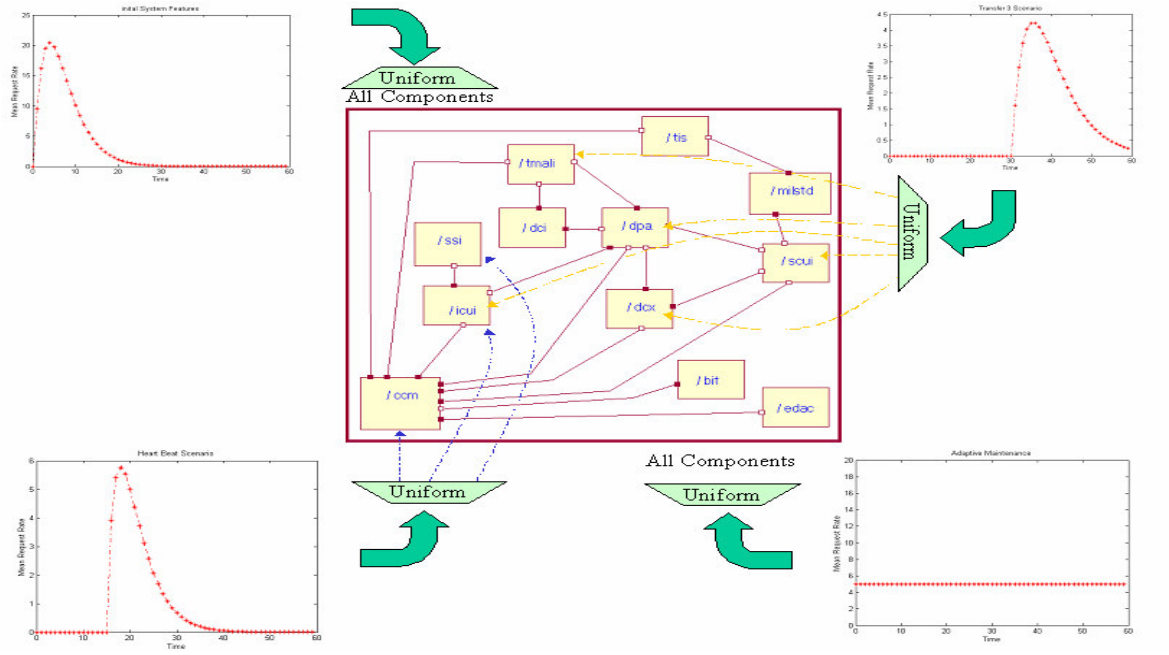


Figure 75 Simulation settings for perfective and adaptive maintenance for CM1

Figure 76 shows the estimated mean request arrivals rate for maintenance simulation of CM1 generated from the simulation following the mentioned settings. The estimated maintenance requests per component using the simulation are shown in Figure 77. To get an estimate for the initial change probability shown in Figure 78, we normalize these estimated maintenance requests per component by the max sum of generated request at any time.

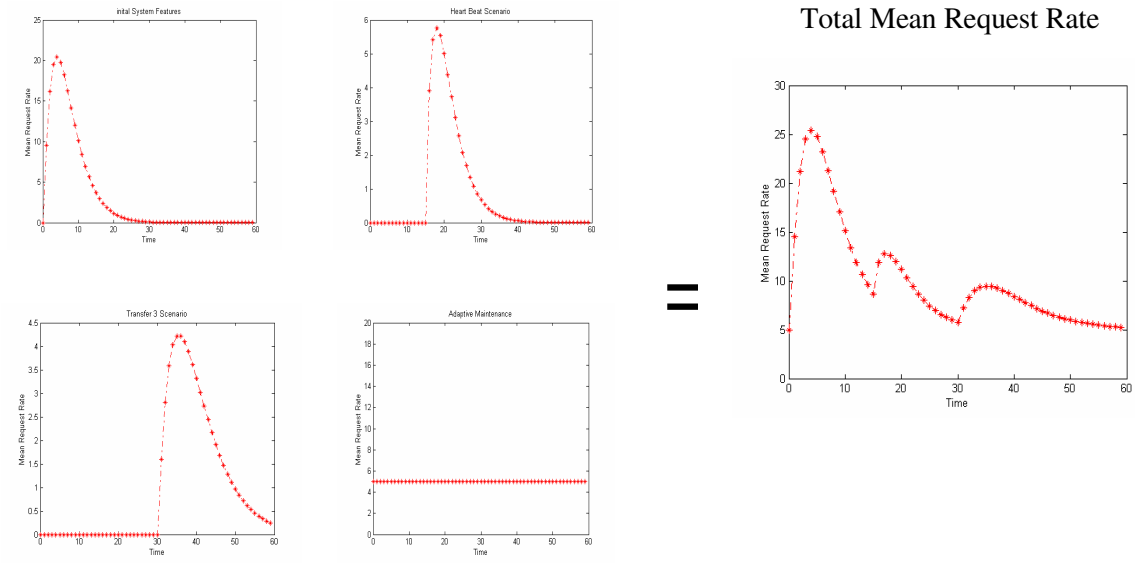


Figure 76 Estimated mean request arrivals rate for maintenance simulation of CM1

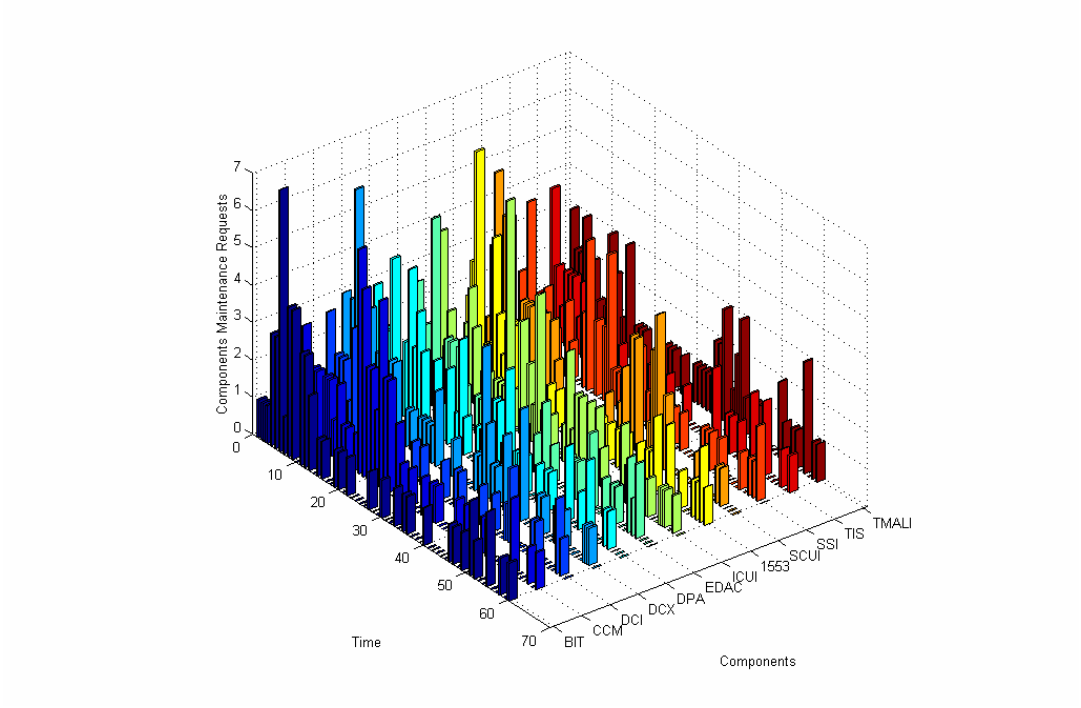


Figure 77 Estimated maintenance requests per component using the simulation

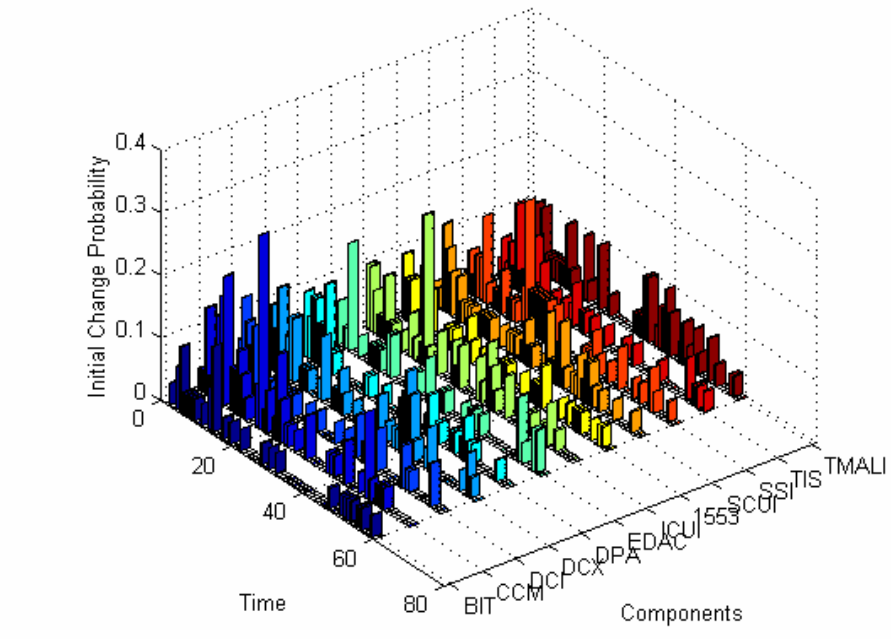


Figure 78 Initial change probabilities for CM1 components

The change propagation probabilities and the size of change for the CM1 case study are given in Figure 53 and Figure 54. Using equation (7.1), the maintainability-based component risk factor for each CM1 component is estimated as a function of time. The components maintainability- based risk for CM1 case study is shown in Figure 79. Across the time of the simulation, we found that the most risky component is CCM. This is due to the fact that CCM is coupled to most of the components of the system, so it is likely to be affected by the changes introduced in these components. Furthermore, CMM has a high maintenance impact on the rest of CM1 components. As it is coupled to other components in the system other than the components of the initial change set, the change is likely to propagate further. Other risky components are ICUI and SCUI. Note that we can locate peaks in the function of the risk level of these risky components as a result of introducing new features into the system.

On the other hand, components like DCX, SSI and TMALI even though they are part of the components affected by the assumed change scenarios; they have low level of maintainability risk because they are coupled to a limited number of components in CM1 case study. Therefore, they have a limited maintenance impact and they are less risky in terms of maintainability.

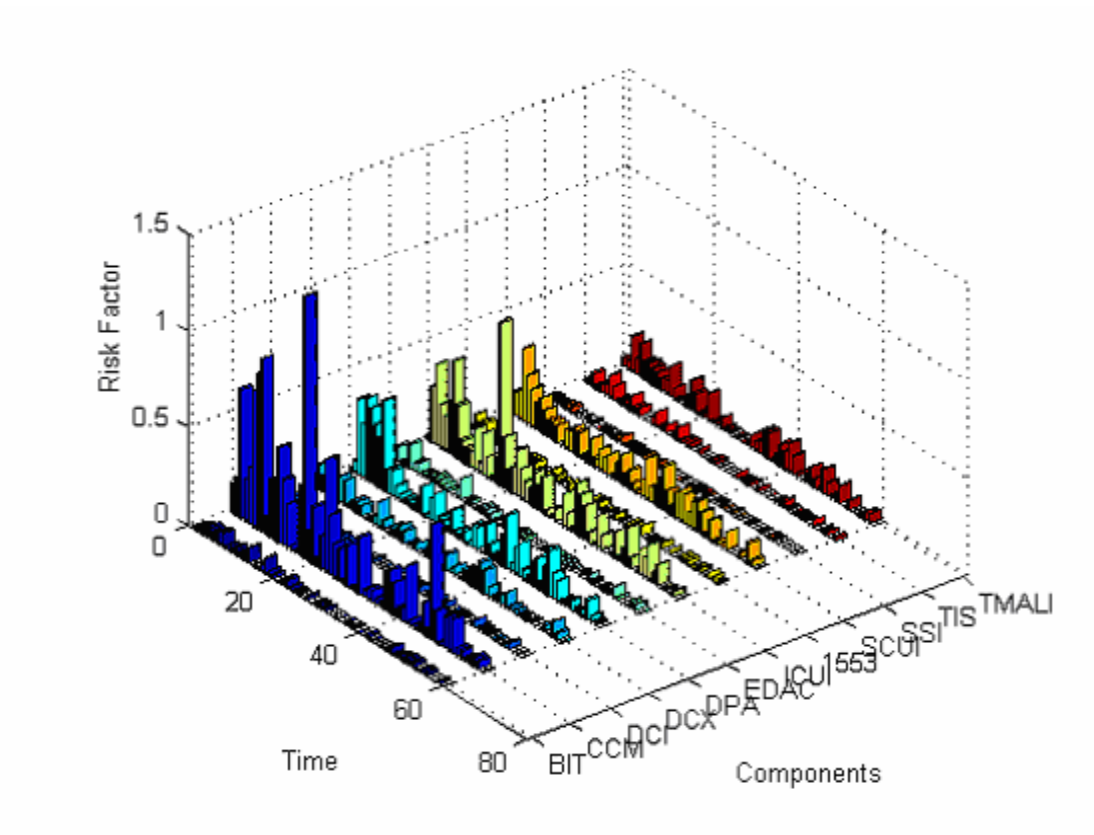


Figure 79 Components maintainability-based risk for CM1 components

7.9 Validation Prospects for Maintainability Based Risk Estimation

In order to demonstrate the value of applying the maintainability-based risk assessment, an experiment design should incorporate monitoring several maintenance projects with different size and try to accomplish the following:

- In a bottom-up approach, we should investigate the validity of the metrics used in the estimation methodology. We need to ensure that the initial change probabilities are predicted correctly. We have to validate that change propagation is accurately estimated and is reflecting on the conditional probability that a change in a component might cause propagation of changes into other components of the system (See section 6.3 and section 6.4). Also, the size of change metrics should be examined to make sure that it correlates with the maintenance impact of the change under consideration.
- For each project, we should conduct an analysis of the system components changes history to evaluate the change impact for each maintenance task and the frequency of changes to estimate the probability of change for the system components.
- In the monitoring process, we should continuously evaluate the maintainability-based risk of the system components and keep track of it and compare it with the subjective assessment of the components maintainability risk by the maintenance engineers with in-depth knowledge of the maintained systems.

  In order to validate the methodology, we carried out some pilot exploratory studies:

  - We used the architectural artifacts and maintenance data for the CM1 case study from NASA Metrics Data Program MDP. We applied the estimation methodology for the maintainability-based risk assessment on the system for different type of software maintenance (See section 7.3.1, section 7.4.1, section 7.5.1 and section 7.8). As future validation work we intend to contact a domain expert for the CM1 case study and use their subjective judgment to assess our results and check whether they reflect the level of maintainability risk of the system components.
  - We executed controlled experiment on three case studies: Job Application, Colleague states and Borg (See section 6.6 and section 7.6). We performed pre/post analysis of software systems for different types of maintenance activities in a controlled and managed environment, i.e. we know the set of changes that will be applied and the objective of making the change (e.g. refactoring part of the system or applying some design patterns to improve the architecture of the system). Then, we examined the components maintainability risk to see if the changes in their risk level

can be explained in accordance with the maintenance objective, i.e. they reflect the objectives and the intentions required from the maintenance task.

7.10 Summary and Discussion

In this chapter, we defined maintainability-based risk as a product of two factors: the probability of performing maintenance tasks and the impact of performing these tasks. We presented a generic methodology for assessing maintainability-based risk to account for changes in the system components in the context of corrective, adaptive and perfective maintenance. The proposed methodology depends on the architectural artifacts and their evolution through the life cycle of the system. We illustrated the methodology on three case studies using UML models. One of these cases studies is an industrial real software system; we plan to validate our results obtained with subjective assessment of the system from its developers and maintainers.

In order to validate the maintainability-based risk assessment methodology, we need to track a system form the early life-cycle stages and continuously assess the components maintainability-based risk through the development stages. Finally, we asses the maintainability of the software product and compare it with the earlier results. We need to notice that beside the resources and time span required for such a research, there is a risk that the system developed early in the life-cycle will deviate significantly forms the original models that we based our estimates upon. We would end up with comparing two different systems. This would result form many factors that you can not control.

Even though, it is important to notice that the modular approach of our maintainability-based risk assessment allow flexible improvements and validation of specific modules in our proposed methodology. Thus, we can easily modify the risk estimation according to the finding of the empirical maintenance data in order to have a better assessment for the component maintainability risk. Furthermore, the modular facilitates validation of parts of the methodology. As we validated the estimation of change propagation probabilities, we plan to validate the other modules in the approach.

For example, we used the size of change as way to predict the maintenance impact. We need to refine this module in the methodology to capture not only the size of changes introduced in the component but also to take into account the effort of finding these changes and conducting these changes. For example, we can use the cyclomatic complexity of the component combined with the size of change. In this case, we assume that the more complex a component is, more effort is required to find where to introduce the changes and more difficult to carry out these changes. Such enhancements will affect only some module in the methodology without altering the whole structure.

8 Software Architecture Risk Assessment (SARA) Tool

In this chapter, we describe the Software Architecture Risk Assessment (SARA) tool that support architectural level model-based risk assessment [Sheik 2006]. The SARA tool provides estimates for reliability-based risk, requirements-based risk and maintainability-based risk. The tool extends our earlier Architectural-level Risk Assessment Tool [Wang+ 2003] by providing support for more architectural models and different perspective of risk assessment other than reliability-based risk.

8.1 Structural Description

The architecture of SARA tool is shown in Figure 80. It accepts different input formats, such as Rose RealTime [Rational Rose RT] models, StarUML [StarUML] models and Java Understand [JavaUnderstand] static analysis files.

First, we extract the required architectural –level information from the examination of these inputs and store it in the software architecture artifacts repository for further analysis. According to the type of risk assessment to be performed, the tool evaluates the metrics required such as cyclomatic complexity, dynamic coupling, change propagation probabilities, size of change and error propagation probabilities using the stored artifacts in the repository.

Then, the tool admits the analyst to provide the severity analysis corresponding to the considered type of risk. Finally, the tool provides the analyst with the risk estimates for the components of the system.

8.2 Functional Description

In our functional description of the SARA tool, we concentrate on the maintainability-based risk assessment part of the tool. For the details of the functional description for the other functionalities check [Wang 2003]. The tool enables automatic assessment of the risk and hence makes it possible for the analyst to identify critical components. The tool automates the steps of the estimation methodology for the maintainability-based risk. The tool estimates change propagation probabilities, and size of change metrics collected from architectural information of the system. The output of the tool can help in the allocation and management of the maintenance effort.

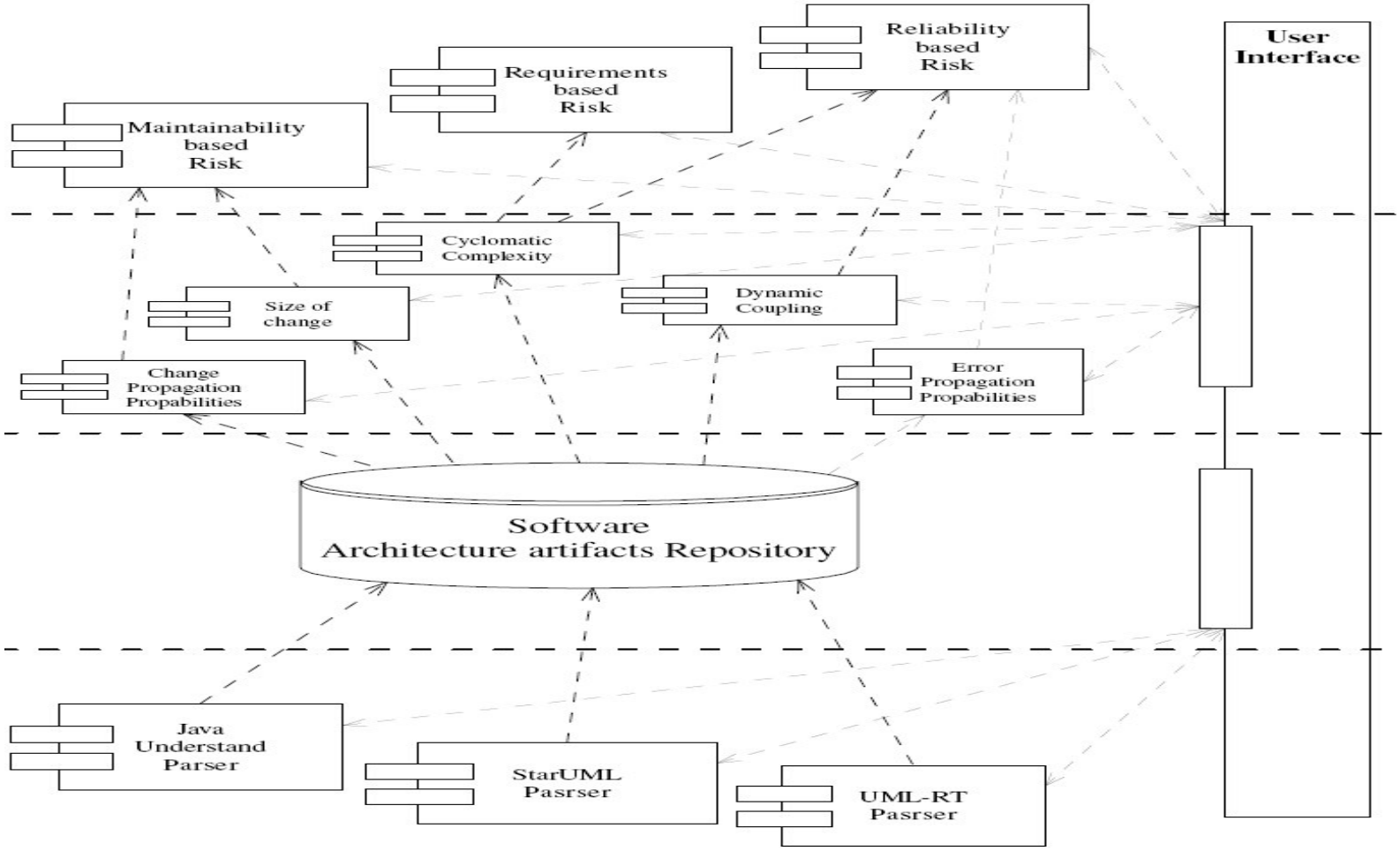


Figure 80 The architecture of the Software Architecture Risk Assessment (SARA) Tool

Our methodology for estimating maintainability-based risk depends on architectural artifacts collected by static analysis of the source code files of the system. First, we estimate the initial change probabilities using metrics reflecting the bad smells of the components of the software architecture. Using the initial change probabilities of components and change propagation probabilities between them, we get the unconditional probability of change of the components of the system. To get the impact of the maintenance tasks, we use the size of change between the components of the system. Finally, the maintainability-based component risk factor is estimated using equation (7.1). In Figure 81, a UML use case model of the tool is shown.

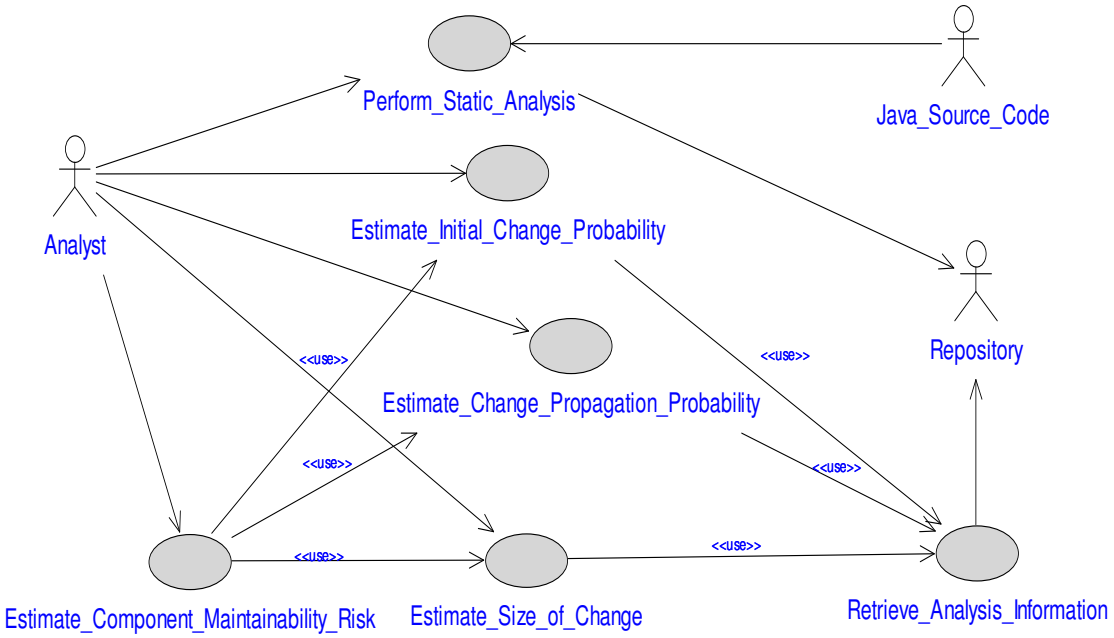


Figure 81 Use case diagram of maintainability-based risk functionality of the SARA tool

In Figure 82, a snap shot of the tool is showing the results of change propagation probabilities obtained from a StarUML model. In Figure 83, a snap shot of the tool showing the results of the maintainability-based risk for corrective maintenance of the same case study.

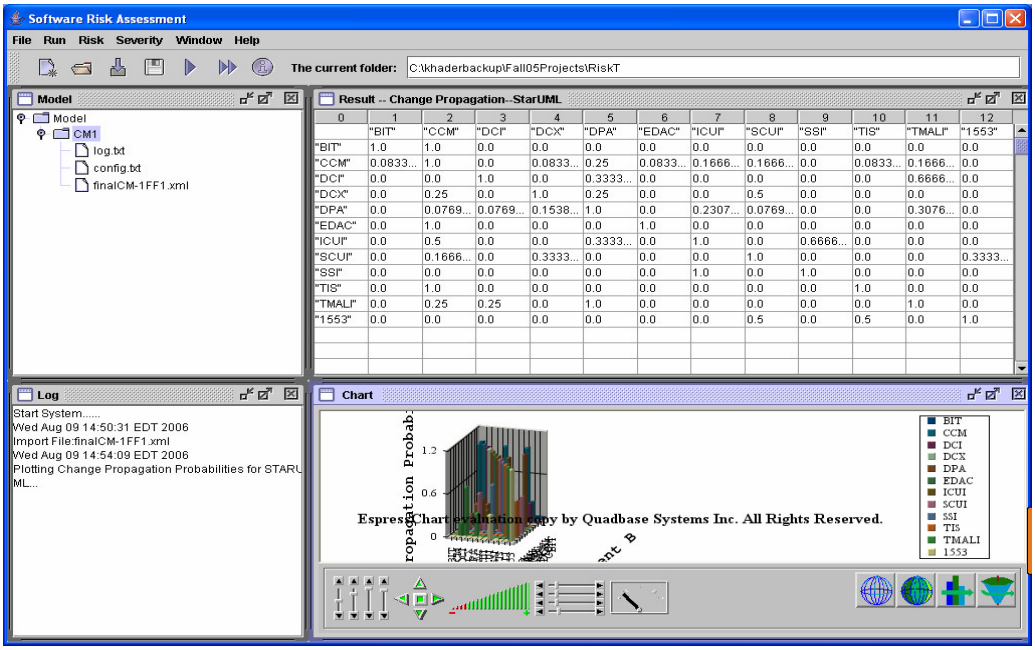


Figure 82 Change propagation probabilities for StarUML model

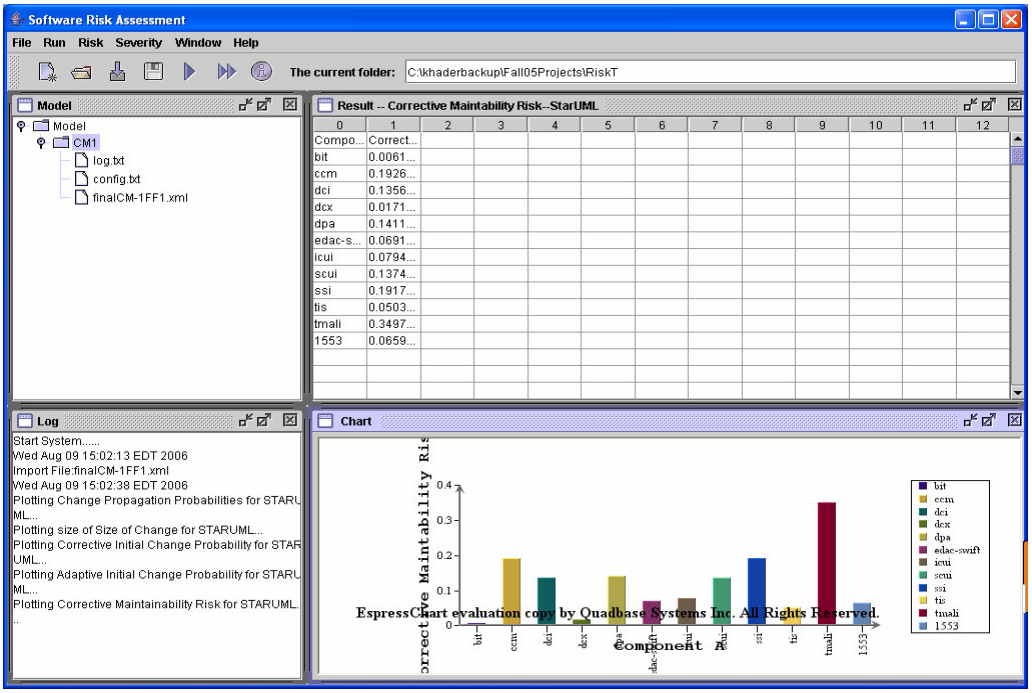


Figure 83 Maintainability based risk for corrective maintenance

9 Conclusions and Future Work

In this dissertation, we present risk assessment estimation methodologies based on the architectural models of the software system. These methodologies rely on quantitative assessment of non-functional quality attributes rather that subjective predicators based on domain expert knowledge. These subjective predictors are human intensive and error prone only. Moreover, there is a need to shift the emphasis of metrics from design and code artifacts into new metrics that address the risk of making requirement changes and improvements of the system. It is not easy to deal with these kinds of artifacts that relate to requirements changes and perfective maintenance. Even though, it is beneficial to have early indicator of future software problems. The deliverables of this dissertation as new contributions can be arranged into:

1. Reliability-based risk assessment:

   - Used Error Propagation Probability in the reliability-based risk assessment to account for the dependency among the system components. We conducted an analytical and an experimental analysis of error propagation probabilities for a command and control system case study. Also, we correlated the analytical and empirical results. Furthermore, we addressed the generalization for the assumption of error occurrence independence in the components of the system by accounting for error propagation among the components of the system on a number of case studies.

   - Generalized the reliability-based risk assessment to account for functional dependencies. In the context of object-oriented Unified Modeling Language UML, we handled risk assessment with use-case relationships. We first proposed a method which is used to estimate the risk factor of a non-primitive use case related to a primitive use case by either <<extend>> or <<include>> relationship. Then, we proposed an algorithm that allows us to estimate the use cases and system risk factors from a general use case diagram that may include many different <<extend>> and <<include>> relationships among use cases. Finally, we applied the generalized reliability-based risk assessment methodology on an industrial case study. It should be emphasized that although many papers used use cases for different type of quantitative analysis, to the best of our knowledge, this is the first work which accounts for relationships between use cases.

2. Maintainability-based risk assessment:

   - Introduced and defined maintainability-based risk, which assesses how difficult it is to maintain the system in the future because of possible maintenance task, as a product of two factors: the

probability of performing maintenance tasks and the impact of performing these tasks. We investigated the maintainability risk of the system components, and the effect of performing the maintenance tasks.

- Developed a general methodology for estimating the maintainability-based risk when considering different types of maintenance that the software undergoes such as adaptive, corrective and perfective maintenance. The proposed methodology depends on the architectural artifacts and their evolution through the life cycle of the system. We applied the proposed methodology on several case studies.
- Automated the estimation of the maintainability-based risk assessment methodology. We presented the architecture of the Software Architecture Risk Assessment (SARA) tool. We extended the functionality of the by providing support for more architectural models and different perspective of risk assessment other than reliability-based risk, specifically maintainability based risk assessment

The future work has the following aspects:

1. We plan to apply the generalized reliability-based risk assessment methodology presented in this dissertation on other case studies.
2. To validate the maintainability-based risk assessment estimation methodology, we also intend to apply methodology on other case studies/system architectures such as product lines and evaluate the estimated risk against actual maintenance records.
    a. We intend to mine history repositories of the open-source projects to analyze its change data and use it to give insights about the maintainability risk of the systems. Then, we can compare it with our estimation procedure results. Moreover, we can contact the administrators of such projects to get a subjective assessment of the maintainability of the components of the system.
    b. We will execute additional controlled experiments and perform pre/post analysis of software systems for different types of maintenance activities in a controlled and managed environment.
    c. We intend to further refine the model by using better estimators for the parameters. For example to better estimate the maintenance impact, size of change parameter could be weighted by the complexity of the component to capture the difficulty of finding and

making changes. In that case, we assume that it is more difficult to find where changes needed to be conducted and to perform these changes in a more complex component than the case in a less complex one. Also, incorporating multi-step change propagation in the model would get a better estimate for components maintainability-based risk.

3. In our risk assessment methodology, we used the structure of the system as an equivalent to the architecture without considering the different architecture styles that could be adopted. For the purpose of our analysis, we only need the control flow and component interactions. We need to refine our methodologies to take into consideration the different styles of the software architecture and how they would affect the risk factors estimation.

4. We aim to extend the Software Architecture Risk Assessment (SARA) tool so that it can account for the changes in the system requirements and to support the generalized reliability-based risk assessment methodology to account for the use-case relationships.

I. Glossary

Abstraction: A model that summarizes the details of the subject it is representing.

Actor*:* External entities interacting with the design

Adaptive Change: A change made in order to adapt the system to changes in its data environment or processing environment.

Change: The act, process, or result of being made different in some particular.

Class Associations*:* Relationships between classes, which can be aggregation, composition, generalization, and dependency as specified in UML

Corrective Change: A change made in order to correct processing, performance, or implementation failures of the system.

Dependency: In general, a dependency implies that the complete functioning of an element requires the presence of another, which exists in the same level of abstraction or realization (i.e. pattern, class, or subsystem level of abstraction).

Design Pattern: A design component composed of collaborating classes that are customized to solve a general frequent-recurring design problem in a particular context.

Empirical: Capable of being verified or disproved by observation or experiment.

Environment: The totality of conditions and influences which act from outside upon an entity.

Evolution: A process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state.

Forward Engineering: The traditional software engineering approach starting with requirements analysis and progressing to implementation of a system.

Framework: A set of ideas, conditions, or assumptions that determine how something will be approached, perceived, or understood.

Impact Analysis: The determination of the major effects of a proposed project or change.

Maintainability: The ease with which maintenance can be carried out.

Maintenance Personnel: The individuals involved in maintaining a software product.

Maintenance Process: Any activity carried out, or action taken, either by a machine or maintenance personnel during software maintenance.

Maintenance: The act of keeping an entity in an existing state of repair efficiency, or validity; to preserve from failure or decline

I. Glossary

Measurement: The process of empirical, objective encoding of some property of a selected class of entities in a formal system of symbols so as to describe them.

Methodology: is a collection of methods applied across the software development life cycle and unified by some general philosophical approach.

Metric: A criterion to determine the difference or distance between two entities, like the distance of a query and a document in information system Retrieval systems.

Object Oriented Programming: computer programming in which code and data pertaining to a single entity (object) are encapsulated, and communicate with the rest of the system via messages.

Operating Environment: All software and hardware systems that influence or act upon a software product in any way.

Perfective Change: A change made in order to perfect the system in terms of its performance, processing efficiency, or maintainability.

Preventive Change: A change made in order to prevent system problems before they occur.

Product: a concrete documentation or artifact created during a software project.

Program: Code components, at the source and object code level, such as modules, packages, procedures, functions, routines, etc. Also commercial packages such as spreadsheets and databases.

Quality Assurance: The systematic monitoring and evaluation of aspects of a project, service or facility to ensure that necessary standards of excellence are being met.

Reengineering: The process of examination and alteration whereby a system is altered by first reverse engineering and then forward engineering

Restructuring: The transformation of a system from one representational from to another.

Reverse Engineering: The process of analyzing a subject system to:

- Identifying the system's components and their interrelationships and
- Create representations of the system in another form or at higher levels of abstraction

Ripple Effect: Consequences of a action in one place, occurring elsewhere e.g. a stone dropped in a pond resulting in waves/ ripples far from the point of impact.

Safety-Critical: A system where failure could result in death, injury or illness, major economics loss, environmental or property damage.

Software Architecture of a program or computing system is the structure or structures of the system, which comprise software elements the externally visible qualities of those elements, and the relationships among them

Software Evolution: The tendency of software to change over time.

I. Glossary

Software Maintenance Framework: The context and environment in which software maintenance activities are carried out.

Software Maintenance Tool: An artifact used to carry out automatically a function relevant to software change.

Software: The programs, documentation and operating procedures by which computers can be made useful to man.

Tool: Implement or device used to carry out functions automatically or manually.

UML: The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.

II. Bibliography

[Abdelmoez+ 2003] W. Abdelmoez, A. Hassan, A. Guedem, K. Goseva-Popstojanova, H. Ammar, "Considering Use Case Dependencies in Architectural-Level Risk Analysis Based on UML Specifications" *Suppl. Proc. 14th International Symposium on Software Reliability Engineering (ISSRE'03),* November 17 - 20, 2003 Denver CO., pp. 323-324.

[Abdelmoez+ 2004A] W. Abdelmoez, D.M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H.H. Ammar, Bo Yu, A. Mili, "Error Propagation In Software Architectures", Proc. 10th IEEE International Software Metrics Symposium (METRICS 2004), Chicago, IL, September 2004.

[Abdelmoez+ 2004B] W. Abdelmoez, R. Gunnalan, M. Shereshevsky, H.H. Ammar, Bo Yu, M. Korkmaz, A. Mili, "Software Architectures Change Propagation Tool (SACPT)", Proc. 20th IEEE International Conference on Software Maintenance (ICSM 2004), Chicago, IL, September 2004.

[Abdelmoez+ 2005A] W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H.H. Ammar, Bo Yu, S. Bogazzi, M. Korkmaz, A. Mili, "Quantifying Software Architectures: An Analysis of Change Propagation Probabilities", ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 05), Cairo, Egypt, January 3-6, 2005.

[Abdelmoez+ 2005B] W. AbdelMoez, I. Shaik, R. Gunnalan, M. Shereshevsky, K. Goseva-Popstojanova, H.H. Ammar, A. Mili, C. Fuhrman, "Architectural level Maintainability Based Risk Assessment", Proc. of poster papers in IEEE International Conference on Software Maintenance (ICSM 2005), Budapest, Hungray, September 25-30,2005.

[Abdelmoez+ 2006A] W. Abdelmoez, K. Goseva-Popstojanova, H.H. Ammar," Methodology for Maintainability-Based Risk Assessment", Proc. of the 52nd Annual Reliability & Maintainability Symposium (RAMS 2006), Newport Beach, Ca., January 23-26, 2006.

[Abdelmoez+ 2006B] W AbdelMoez, K. Goseva-Popstojanova, H.H. Ammar, "Maintainability-Based Risk Assessment in Adaptive Maintenance Context", Proc. of the 2nd PRedictOr Models In Software Engineering (PROMISE 2006) workshop, Philadelphia, Pa. USA, September 24, 2006.

[Abdelmoez+ 2006C] W. AbdelMoez, K. Goseva-Popstojanova, H. Ammar, "Using Maintainability Based Risk Assessment and Severity Analysis in Prioritizing Corrective Maintenance Tasks",

Suppl. Proc. 17th International Symposium on Software Reliability Engineering (ISSRE'06), Raleigh, NC., November 7-10, 2006.

[Alexander 1979] Alexander, C., "The Timeless Way of Building", Oxford University Press, 1979.

[Alexander+ 1977] Alexander, C., S. Inshikawa, M. Silverstiein, M. Jacobson, I. Fiksdahl-king, and S. Angel. "A Pattern Language", Oxford University Press, New York, 1977.

[Ammar+ 2001] H. Ammar, S. M. Yacoub, A. Ibrahim, "A Fault Model for Fault Injection Analysis of Dynamic UML Specifications," International Symposium on Software Reliability Engineering, IEEE Computer Society, November 2001.

[Anderson+ 2002] S. Anderson, M. Felici, "Quantitative Aspects of Requirements Evolution". In Proceedings of the 26th Annual International Conference on Computer Software and Applications Conference, COMPSAC 2002, Oxford, England, 26-29th August 2002, IEEE Computer Society, pp. 27-32.

[Armour+ 2001] F. Armour and G. Miller, *Advanced Use Case Modeling*, Addison-Wesley, 2001.

[Baude 2003] E. Baude, "Software Design: From Programming to Architecture", Wiley, 2003.

[Beck+ 1994] Beck, K., R. Johnson, "Patterns Generate Architectures" ECOOP'94, LNCS 821, pp139-149.

[Beck+ 1996] Kent Beck, James Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, John M. Vlissides: Industrial Experience with Design Patterns. ICSE 1996: 103-11

[Bohner+ 1996] Bohner, S.A., Arnold, R.S. Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[Booch+1999] Grady Booch, Jim Rumbaugh, and Ivar Jacobson, "The Unified Modeling Language User Guide", ISBN: 0-201-57168-4, Addison Wesley, est. publication December 1997.

[Borg] Source Forge Project: BORG Calendar http://sourceforge.net/projects/borg-calendar/

[Bosch+ 2001] J. Bosch and P. Bengtsson, "Assessing Optimal Software Architecture Maintainability", Proc. of fifth European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, March 2001.

[Bowles 1998] J. Bowles, "The New SEA FMECA Standard", Proc.1998 Annual Reliability and Maintainability Symp. (RAMS 1998), Anaheim, California, 1998, pp. 48-53.

II. Bibliography

[Briand+ 1999A] Briand L., Wust J, Ikonomomovski S. and Lounis H, “Investigating Quality Factors in Object Oriented Designs: An Industrial Case Study,” Proc. of the 1999 International Conference on Software Engineering, Los Angeles, May 16-22, 1999,pp 345-354.

[Briand+ 1999B] Briand L., J. Wuest, H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented System", IEEE International Conference on Software Maintenance (ICSM), 1999, Oxford, UK.

[Briand+ 2003] Briand L., Labiche Y., O'Sullivan, “Impact Analysis and Change Management of UML Models”, IEEE International Conference on Software Maintenance (ICSM) 2003.

[Burch+ 1997] Burch E. and H. Kung, “Modeling Software Maintenance Requests: A Case Study,” Proc. IEEE Int’l Conf. Software Maintenance, pp. 40-47, 1997

[Buschmann+ 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, "Pattern-Oriented Software Architecture - A Pattern System", Addison-Wesley, 1996.

[Canning 1972] Canning RG, “That maintenance iceberg”, *EDP Analyzer* 1972,(10):1–14.

[Cantone+ 2004] G. Cantone, D. Pace, G. Calavaro, “Applying Function Point to Unified Modeling Language: Conversion Model and Pilot Study”, Proc. of 10th International Symposium on (METRICS'04), September 11 - 17, 2004, Chicago, Illinois, pp.280-291.

[Card+ 1990] Card, D.N. and Glass R.L., Measuring Software Design Quality, Prentice Hall, Englewood Cliffs, New Jersey,1990.

[Chapin 2000] Chapin N., “Do we know what preventive maintenance is?”, In *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000.

[Cheung 1980] Cheung R. C., “A User-Oriented Software Reliability Model”, IEEE Transactions on Software Engineering, Vol.6, No.2, 1980, pp. 118-125.

[Chidamber+ 1994] Chidamber S.M and Kemerer C.F, “A Metrics Suite for Object Oriented Design,” IEEE Transactions on Software Engineering, Jun 1994, pp 476-493.

[Clarkson+ 2000] Clarkson, P.J., Simons, C. and Eckert, C.M., “Change propagation in the design of complex products”, in Engineering Design Conference (EDC2000), Brunel University, Uxbridge,2000, 563-570

[Clarkson+ 2001] Clarkson, P.J., Simons, C. and Eckert, C.M., “Predicting change propagation in complex design”, Proceedings 13th International Conference on Design Theory and Methodology

(DETC'01), ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Pittsburgh, Pennsylvania, USA, 2001.

[Cohen+ 2000] Cohen T., Navthe, S. and Fulton, R. E.,"C-FAR, change favorable representation", Computer-Aided Design 32: 321-38, 2000.

[Costello 2005] Costello K., Software Integrity Level Assessment Process (SILAP), NASA IV&V Facility, 2005.

[Cunningham 1994] Cunningham, W., "The CHECKS Pattern Language of Information Integrity", in Proceedings of Pattern Languages of Program PLoP'94

[Douglass 1998] Douglass B., Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1998.

[Eppinger+ 1994] Eppinger, S. D., Whitney, D. E., Smith, R. P. and Gebala, D. A.,"A Model-based Method for Organizing Tasks in Product Development", Research in Engineering Design 6(1):1-13,1994.

[Fanta+ 1998] Fanta, R., Rajlich, V. Reengineering an Object Oriented Code. In Proceedings of IEEE International Conference on Software Maintenance,1998, IEEE Computer Society Press, 238-246

[Fenton+ 1996] Fenton, N.E. and Pfleeger, S.L., Software Metrics, 2nd edition, Thomson Publishing Inc., 1996.

[Fenton+ 2000] Fenton, N.E. and Ohlsson, N., " Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Trans Software Engineering,* Vol. 26, No. 8, pp. 797-814.

[Fowler+ 1999] Fowler M. and Beck K, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[Gamma 1991] E. Gamma, "Objectorientierte Software-Entwicklung am Beispiel von ET++" Klassenbibliothek, Werkseuge, Design, Dissertation, Universitat Zurich, 1991 (*Translated as* "Object-oriented software development with examples from ET++: class library, tool, and design").

[Gamma+ 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Object-Oriented Software", Addison-Wesley, 1995.

II. Bibliography

[Gefen+ 1996] Gefen D. and S.L. Scheberger, "The NonHomogeneous Maintenance Periods: A Case Study of Software Modifications," Proc. 1996 Int'l Conf. Software Maintenance (ICSM '96), pp. 134-141, 1996.

[Gill+ 1991] G.K. Gill, C.F. Kemerer, "Cyclomatic Complexity Density and Software Maintenance Productivity," *IEEE Trans. Software Eng.*, Vol.17,No. 12, pp. 1284-1288, 1991.

[Goseva-Popstojanova+2001] K. Goseva-Popstojanova and K. S. Trivedi, "Architecture Based Approach to Reliability Assessment of Software Systems", Performance Evaluation, Vol. 45, No. 2-3, 2001, pp. 179-204.

[Goseva-Popstojanova+2003] K. Goseva-Popstojanova , A. Hassan, A. Guedem, W. Abdelmoez, D. Nassar, H. Ammar, A. Mili, "Architectural-Level Risk Analysis using UML", *IEEE Trans. Software Engineering,* Vol. 29, No.10, October 2003, pp. 946-960 .

[Grady 1994]. R.B. Grady, "Successfully Applying Software Metrics," *IEEE Computer*, Vol. 27, No. 9, September 1994, pp. 18 - 25.

[Grady+ 1987] R.B. Grady and D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[Guedem 2004] Guedem A. R., "Software Architectural Risk Assessment", Master's Thesis, West Virginia University, 2004.

[Halstead 1977] Halstead, Maurice H., "Elements of Software Science"*, Operating, and Programming Systems Series,* Volume 7. New York, NY: Elsevier, 1977.

[Harel88] David Harel, "On Visual Formalism", *Communications of the ACM*, Vol 31, No 5, May 1988

[Hassan 2004] A. Hassan, "Architectural Level Risk Assessment", Dissertation, West Virginia University, 2004.

[Hassan+ 2001] A. Hassan, W. Abdelmoez, R. Elnaggar, and H. Ammar, "An Approach to Measure the Quality of Software Designs from UML Specifications", Proc. 5th World Multi-Conference on Systems, Cybernetics and Informatics and the 7th Int'l Conf. Information Systems, Analysis and Synthesis, July. 2001, Vol. IV, pp 559-564.

[Hassan+ 2004] A. Hassan and Richard C. Holt, "Predicting Change Propagation in Software Systems", in proceedings of International Conference on Software Maintenance (ICSM) 2004, Chicago, Illinois, USA, September 11-17, 2004

II. Bibliography

[Hawkins+ 2002] R. D. Hawkins and J. A. McDermid. "Performing Hazard and Safety Analysis of Object Oriented Systems", *Proc. 20th International System Safety Conference (ISSC)*, 2002,Denver, CO.

[Hiller+ 2001] M. Hiller, A. Jhumka, and N. Suri, "An Approach for Analyzing the Propagation of Data Errors in Software," *Dependable Systems and Networks*, pp. 161 -170, 2001.

[Houmb+ 2002] S. Houmb, F. den Braber, M. S. Lund and K. Stolen, "Towards a UML Profile for Model-based Risk Assessment", *Proc. UML'2002, Satellite Workshop on Critical Systems Development with UML*, September 30 - October 4, 2002, Dresden, Germany, pp.79-92.

[IEEE 1990] *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1990.

[IEEE 1998] *IEEE Standard for Software Maintenance*, The Institute of Electrical and Electronics Engineers, Inc., New York, 1998.

[IEEE Std 982.1] IEEE Std 982.1- IEEE Standard Dictionary of Measures to Produce Reliable Software.

[Jacobson+ 1992] Jacobson, I., Christerson, M., Jonsson, P., and Overgaard, G., "Object Oriented Software Engineering", Workingham, England: Addison-Wesley 1992.

[JavaUnderstand] Java Understand http://www.scitools.com/uj.html

[Johannessen+ 2001] Johannessen P., Grante C., Alminger A. and Torin U. E. J., "Hazard Analysis in Object Oriented Design of Dependable Systems", Proceeding of the 2001 Int'l Conference on Dependable Systems and Networks, Goteborg, Sweden, July 2001, pp 507-512.

[Kafura+ 1987] Kafura, D.; Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance", IEEE Transactions on Software Engineering, Vol. 13, No. 3, 1987, pp. 335-343

[Kerievsky 2004] Kerievsky J., Refactoring to Patterns, Addison-Wesley, 2004.

[Lazowska 1984] Lazowska E., Quantitative System Performance: Computer System Analysis Using Queuing Network Models, Prentice Hall, 1984.

[Lientz+ 1980] Lientz BP and Swanson EB, "*Software Maintenance Management",* Addison-Wesley Publishing Co.: Reading MA, 1980; 214 pp.

[McCabe 1976] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308-320, 1976.

II. Bibliography

[Menzies+ 2000] Menzies T. and Cukic B., "Maintaining maintainability = recognizing reachability", In International Workshop on Empirical Studies of Software Maintenance (WESS 2000), October 14, San Jose CA, 2000.

[Michael+1997] Michael C. C., and Jones R. C., "On the Uniformity of Error Propagation in Software," *Proc. of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, pp. 68-76, 1997.

[MIL_STD_1629A] Procedures for Performing Failure Mode Effects and Criticality Analysis, US MIL_STD_1629 Nov. 1974, US MIL_STD_1629A Nov. 1980, US MIL_STD_1629A/Notice 2, Nov. 1984.

[Mira 2001] Mira Kajko-Mattsson, "Can We Learn Anything from Hardware Preventive Maintenance?", Proceedings of the Seventh International Conference on Engineering of Complex Computer Systems (ICECCS'01), 2001.

[Moore+ 2003] D.S. Moore and G.P.McCabe, "Introduction to the practice of statistics", W.H. Freeman and Company, 4th edition ,2003.

[Munson+ 1996] J. Munson and T. Khoshgoftaar, "Sotware Metrics for Reliability Assessment," *Handbook of Software Reliability Eng*., M. Lyu, ed., 1996, pp. 493-529

[NASA 1997] NASA-STD-8719.13A,"Software Safety NASA Technical Standard", 1997. http://satc.gsfc.nasa.gov/assure/nss8719_13.html

[NASA MDP] Metrics Data Program, NASA IV&V Facility http://mdp.ivv.nasa.gov/

[Oman 1991] Oman, P. HP-MAS: A Tool for Software Maintainability, Software Engineering (#91-08-TR). Moscow, ID: Test Laboratory, University of Idaho, 1991.

[Oman+ 1992] Oman, P. & Hagemeister, J. Construction and Validation of Polynomials for Predicting Software Maintainability (92-01TR). Moscow, ID: Software Engineering Test Lab, University of Idaho, 1992.

[Oman+ 1994] P. Oman, J. Hagemeister, "Constructing and Testing of Polynomials Predicting Software Maintainability", *Journal of Systems and Software* 24, 3 (March 1994), pp. 251-266.

[OMG 2001] Revision Task Force. OMG Unified Modeling Language Specification, Version 1.4. Technical report, Object Management Group (OMG), 2001. OMG document formal/01-09-67. http://www.uml.org/

II. Bibliography

[OMG 2005] Object Management Group (OMG), "Unified Modeling Language: Superstructure, version 2.0- Final Adopted Specification", OMG document formal/ 05-07-04, Available at: http://www.omg.org/cgi-bin/doc?formal/05-07-04. August 2005.

[OMG UML Profile] UML Profile for Schedulability, Performance, and Time, ptc/02-03-02, OMG Adopted Specification, http://www.omg.org.

[Päivi+ 2002] Päivi Kallio and Tuomas Ihme, "Evolution of the Use and Risks of Commercial Software Components", Proceedings of the 28th Euromicro Conference (EUROMICRO'02), 2002.

[Papadopoulos+ 1999] Papadopoulos Y. and McDermid J. A., "Hierarchically Performed Hazard Origin and Propagation Studies", Proceedings of SAFECOMP '99, 18th International Conference on Computer Safety, Reliability and Security, Toulouse France, Lecture Notes in Computer Science, 1698, Springer Verlag, 1999, pp.139-152.

[Papapanagiotakis +1994] Papapanagiotakis G. and Breuer P., "A software maintenance management model based on queueing networks", Journal of Software Maintenance - Research and Practice, vol. 6, no. 1, pp. 73-97, 1994.

[Pelànek 2004] Pelànek R., "Typical Structural Properties of State Spaces", *in Proc. of 11th International SPIN Workshop on Model Checking of Softwar,* April 1-3, Barcelona, Spain, 2004.

[Pigoski 1996] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, John Wiley & sons, 1996.

[Popic+ 2005] Popic, P. Desovski, D. Abdelmoez, W. Cukic, B., "Error Propagation in the Reliability Analysis of Component Based Systems", Proc. of 16th IEEE International Symposium on Software Reliability Engineering ISSRE 2005, Chicago, Illinois., 8-11 Nov.,2005,pp. 53-62

[Pumfrey 1999] Pumfrey D. J., "The Principled Design of Computer System Safety Analyses", PhD thesis, University of York, Department of Computer Science, September 1999.

[Rajlich 2000] Rajlich, V., "Modeling software evolution by evolving interoperation graphs", Annals of Software Engineering 9: 235-248,2000.

[Rajlich+ 2000] Rajlich, V.T., Bennett, K.H. The staged model of the software lifecycle. IEEE Computer, July 2000, 66-71.

[Rajlich+ 2002] Rajlich, V., Prashant G., A Case Study of Unanticipated Incremental Change, In Proceedings of IEEE International Conference on Software Maintenance, 2002, IEEE Computer Society, 2002, 442 – 451.

II. Bibliography

[Rajlich+ 2004] Rajlich, V., Prashant G., "Incremental Change in Object-Oriented Programming " IEEE Software, July/August 2004, Vol. 21, No. 4, pp.62-69

[Rational Rose RT] Rational Rose Real-Time. http://www.rational.com/products/rosert/index.jtmpl

[Rombach 1987] Rombach, H.D., "A Controlled Experiment on the Impact of Software Structure on Maintainability", IEEE Transactions on Software Engineering, Vol. 13, No.3, 1987, pp. 344-354.

[Rumbaugh+ 1997] Jim Rumbaugh, Ivar Jacobson, and Grady Booch, "Unified Modeling Language Reference Manual", ISBN: 0-201-30998-X, Addison Wesley, est. publication December 1997.

[Schach+ 2000] Schach, S. R. and Tomer, A.,"A maintenance-oriented approach to software construction", Journal of Software Maintenance-Research and Practice 12(1): 25-45,2000.

[SEI 2005] http://www.sei.cmu.edu/architecture/sw_architecture.html Software Architecture for Software-Intensive Systems (last visited November 2005)

[Shaik 2006] I. Shaik , W. AbdelMoez, R. Gunnalan, M. Shereshevsky, A. Zeid, H.H. Ammar, A. Mili, C. Fuhrman, "Using Change Propagation Probabilities to Assess Quality Attributes of Software Architectures" , Proc. of The 4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-06), Dubai/Sharjah, UAE, March 8-11, 2006.

[Shaw 1995] Shaw M., "Architectural issues in software reuse: It's not just the functionality, it's the packaging", *In Proceedings Symposium on Software Reusability*, Seattle, WA, April 1995. Association for Computing Machinery.

[Sheik 2006] Sheik K, AbdelMoez W., Ammar H., "Software Architecture Risk Assessment (SARA) Tool", *Suppl. Proc. 17th International Symposium on Software Reliability Engineering* (ISSRE'06), Raleigh, NC., November 7-10, 2006

[Sherer 1997] Sherer S., "Using Risk Analysis to Manage Software Maintenance," *Software Maintenance: Research and Practice,* Vol. 9, 345-364, 1997.

[Singh+ 2001] H. Singh, V. Cortellessa, B. Cukic, E. Gunel and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component Based Systems", *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE'01),* 2001, Hong Kong, China, pp. 12-21.

[Smith 1990] Smith, C.U. , Performance Engineering of Software Systems, SEI Series in Software Engineering, Addison-Wesley, Readings, Mass. 1990.

II. Bibliography

[Smith+ 2002] Smith, C.U. and Williams L.G., Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software, Addison-Wesley, 2002.

[Stark+ 1994] Stark G.E., Kern L.C., and C.V. Vowell, "A Software Metric Set for Program Maintenance Management", *Journal of Systems and Software,* 1994, pp. 239-249.

[StarUML] StarUML - The Open Source UML/MDA Platform http://staruml.sourceforge.net/en/

[Steward 1981] Steward, D. V., "The Design Structure System: A Method for Managing the Design of Complex Systems", IEEE Transactions on Engineering Management, EM-28 (3),1981.

[Sundararajan 1991] C. Sundararajan, Guide to Reliability Engineering, Data, Analysis, Applications, Implementation, and Management, Van Nostrand Reinhold, New York, 1991.

[Swanson 1976] Swanson EB, "The dimensions of maintenance", In *Proceedings 2nd International Conference on Software Engineering*. IEEE Computer Society Press: Long Beach CA, 1976; 492–497.

[Tan+ 2005] Tan Y. and V. S. Mookerjee, "Comparing Uniform and Flexible Policies for Software Maintenance and Replacement", IEEE Trans. Software Eng, Vol. 31, No. 3, 2005, pp. 238 - 255.

[Trivedi 2002] K. S. Trivedi, Probability and Statistics with Reliability, Queuing and Computer Science Applications, 2nd ed., John Wiley & Sons, 2002.

[Tsantalis+ 2005] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, "Predicting the Probability of Change in Object-Oriented Systems," IEEE Transactions on Software Engineering, vol. 31, no. 7, pp. 601-614, Jul., 2005.

[UML 2005] *Unified Modeling Language OMG* Resource Page http://www.uml.org/ (last visited November 2005)

[Voas 1997] J. Voas**,** "Error propagation analysis for COTS system," *Journal of Computing & Control Engineering*, vol. 8, no. 6, pp. 269 –272, Dec. 1997.

[Wang 2003] Wang T., "Architecture-level Risk Assessment Tool Based on UML Specification", Master's Thesis, West Virginia University, 2003.

[Wang+ 2003] T. Wang, A. Hassan, A. Guedem, W. Abdelmoez, K. Goseva-Popstojanova, H. Ammar, "Architectural Level Risk Assessment Tool Based on UML Specifications", 25th International Conference on Software Engineering, Portland, Oregon, 2003.

II. Bibliography

[Welker+ 1995] Welker, Kurt D. & Oman, Paul W. "Software Maintainability Metrics Models in Practice." Crosstalk, Journal of Defense Software Engineering 8, 11 (November/December 1995): 19-23.

[William+ 2002] William C. Chu, Chih-Wei Lu, Chih-Hung Chang, Yeh-Ching Chung, Yueh-Min Huang and Baowen Xu, "Software Maintainability Improvement: Integrating Standards and Models", Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02), Oxford, England, 26-29 August 2002.

[Yacoub 1999] Yacoub S., "Pattern-Oriented Analysis and Design (POAD): A Methodology for Software Development", Dissertation, West Virginia University, 1999.

[Yacoub+ 1999] S. Yacoub, H. Ammar, and T. Robinson, "Dynamic Metrics for Object-Oriented Designs", Proc. 6th Int'l Symp. Software Metrics (Metrics'99), Boca Raton, Florida, 1999, pp 50-61.

[Yacoub+ 2000] S. Yacoub, T. Robinson, and H. Ammar, "A Matrix-Based Approach to Measure Coupling in Object-Oriented Designs", Journal of Object Oriented Programming, vol. 13, no. 7, Nov. 2000, pp. 8-19.

[Yacoub+ 2002] S. Yacoub and H. Ammar, "A Methodology for Architectural-Level Reliability Risk Analysis," IEEE Trans. Software Eng, Vol. 28, No. 6, June 2002, pp. 529-547.

III. Appendix I : Case Studies

A. Command and Control System Case Study

This case study is a command and control system used in a mission-critical application. We present only the analysis of the Internal Thermal Control subsystem. This subsystem is responsible for providing overall management of pumps, as well as performing the necessary monitoring and responding to sensors data. Also, it is responsible for performing automated startup, and controlling the Internal Thermal Control subsystem reconfigurations. During each execution cycle, incoming commands are checked and validated. A failure recovery system detects failure conditions, such as combinations of Pump failures and Shutoff Valve failures, and performs recovery operations in response to detected failures.

The software architecture of this system is shown in Figure 84. The use case diagram of the Internal Thermal Control subsystem is shown in Figure 85. It consists of 5 actors and 11 use cases. Five use cases are named for mode setting (`Setting_1`, `Setting_2`, `Setting_3`, `Setting_4`, and `Setting_5`) and three use cases are named for pump activation retry (`Pump_1_Retry`, `Pump_2_Retry`, and `Retry_Both_Pumps`). These use cases are examples of primitive use cases. The `Failure_Recovery`, `Monitoring`, and `Mode_setting` use cases are examples of non-primitive use cases. `Monitoring` and `Mode_setting` use cases are also terminal use cases since they are connected directly to actors. Figure 86 to Figure 93 show hierarchal state diagrams for the components of the system. Further details about the system are given in [Hassan 2004].

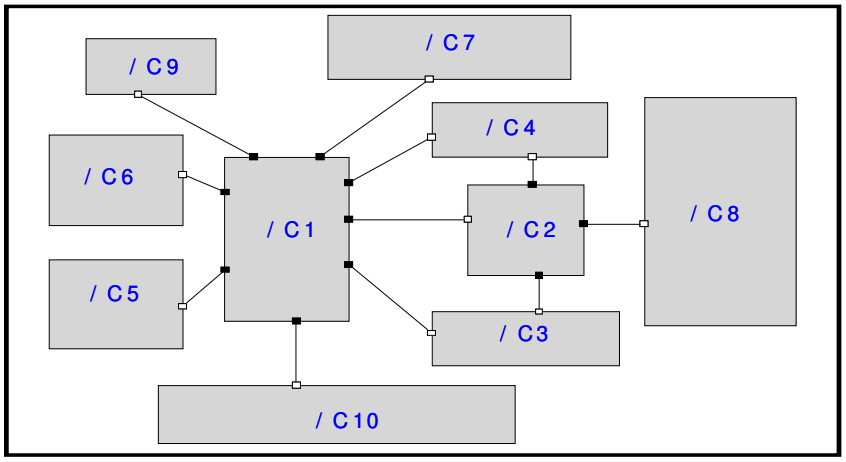


Figure 84 Software architecture of the system.

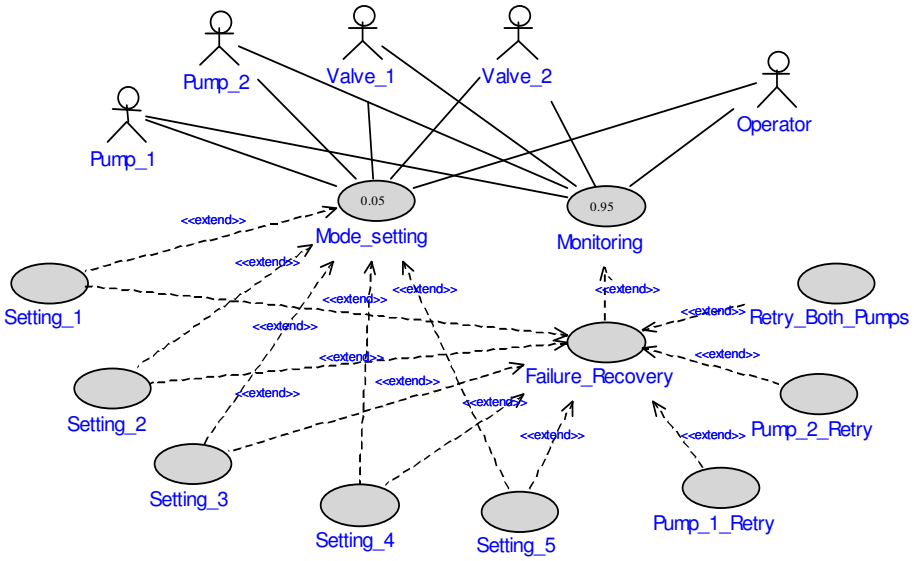


Figure 85 Use case diagram of the Internal Thermal Control subsystem

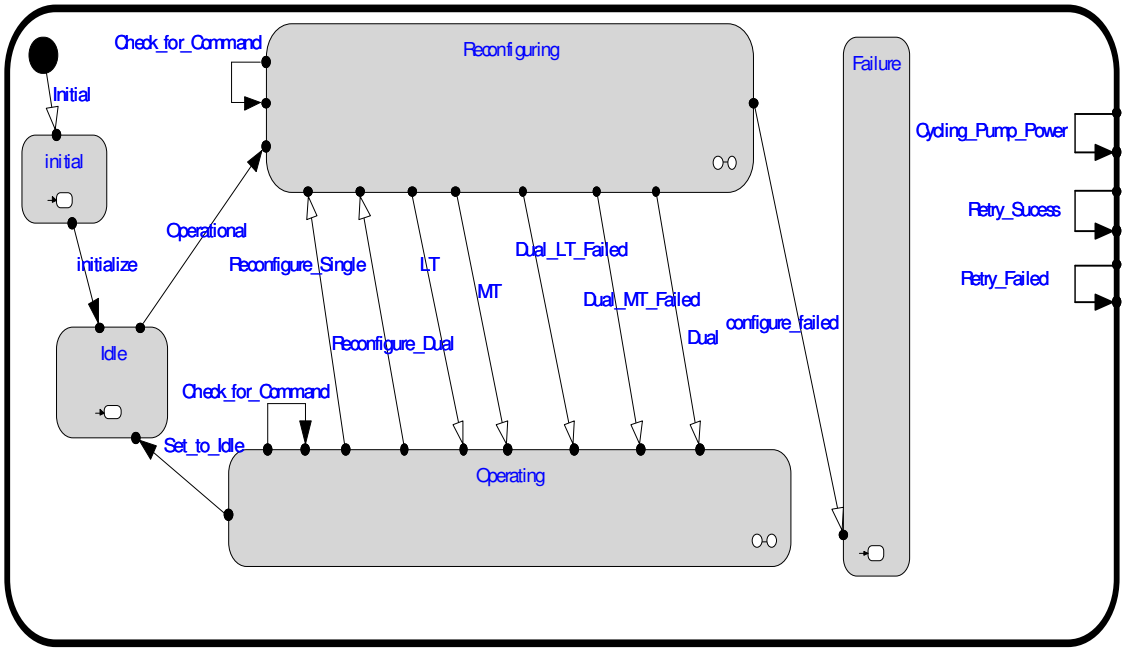


Figure 86 Top-level state diagram of Component C1

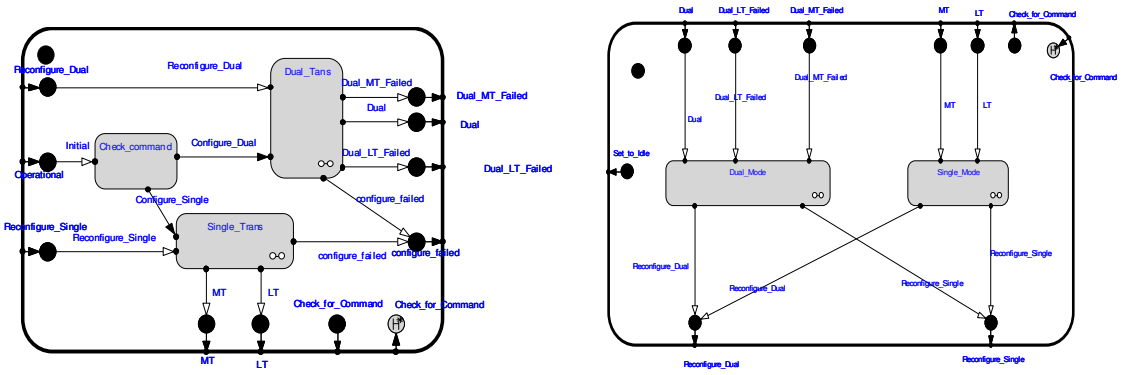


Figure 87 First-level state diagrams of Component C1

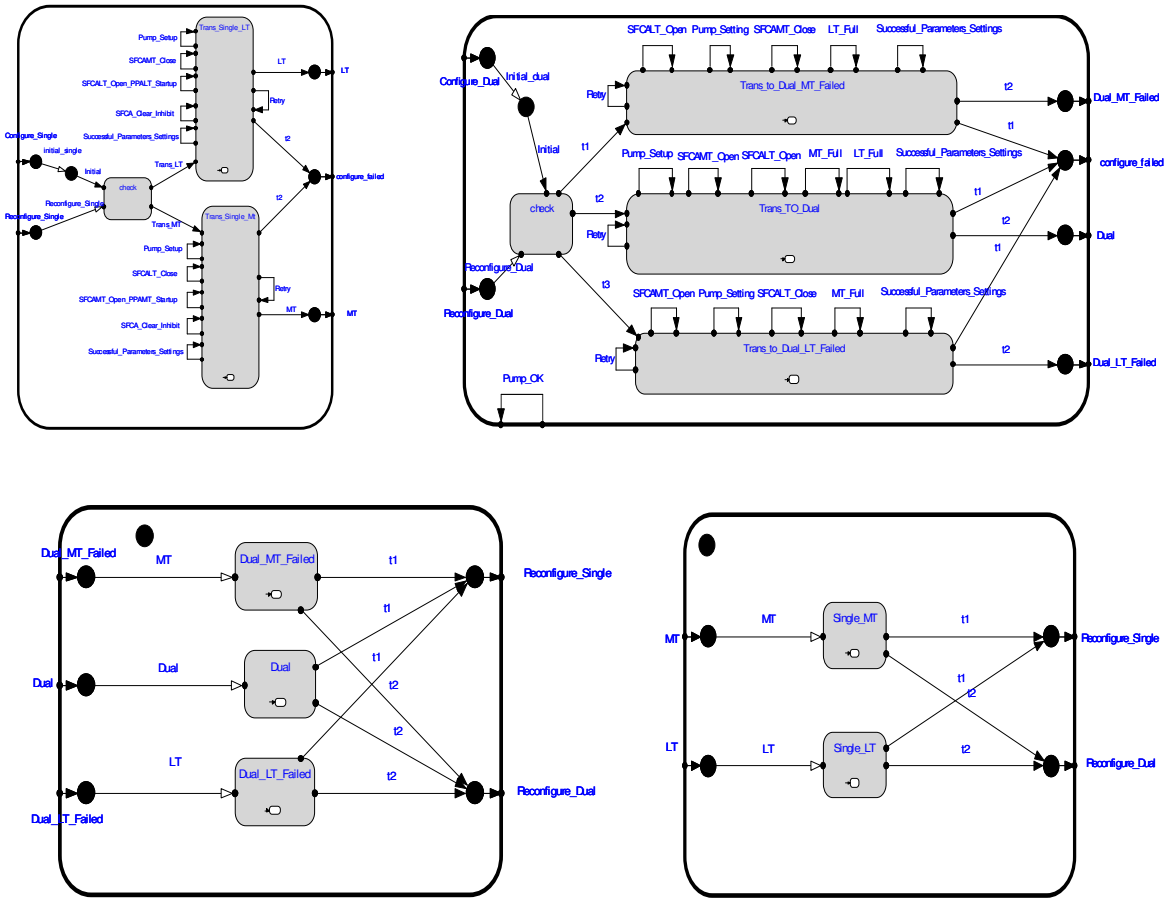


Figure 88 Second-level state diagrams of Component C1

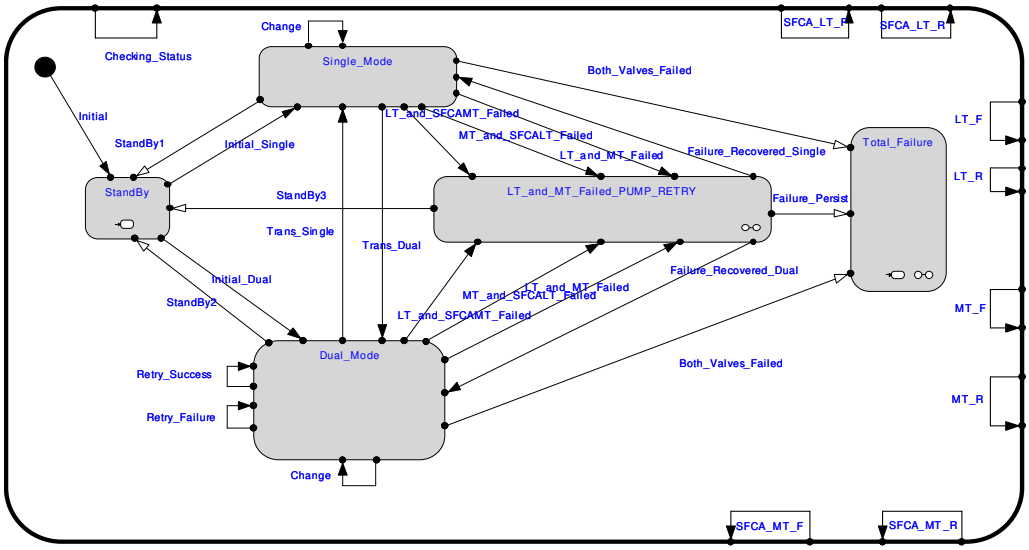


Figure 89 Top-level state diagram of Component C2

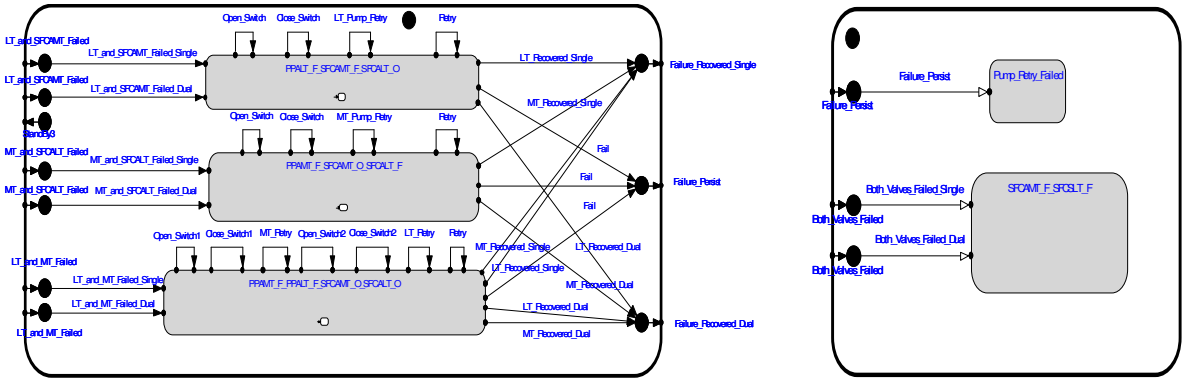


Figure 90 First-level state diagrams of Component C2

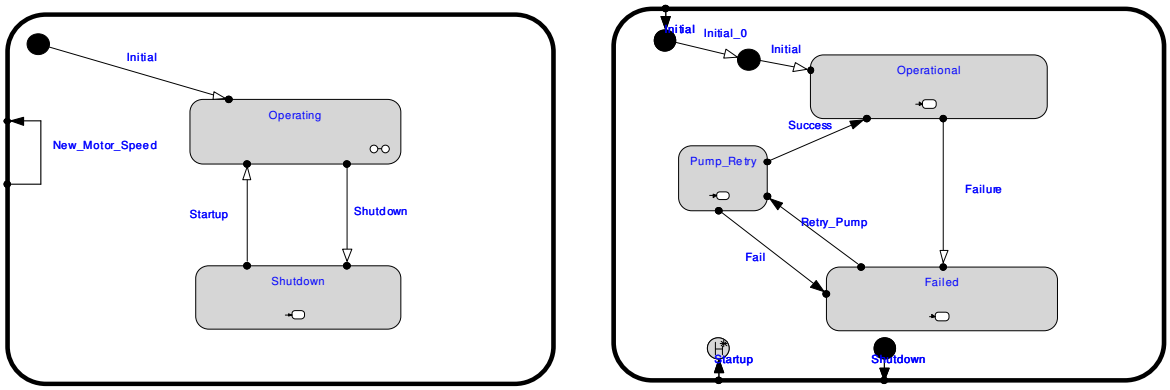


Figure 91 State diagrams of Components C3 and C4

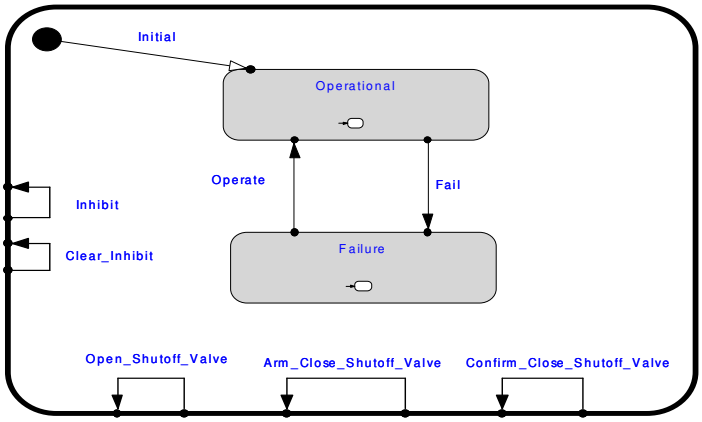
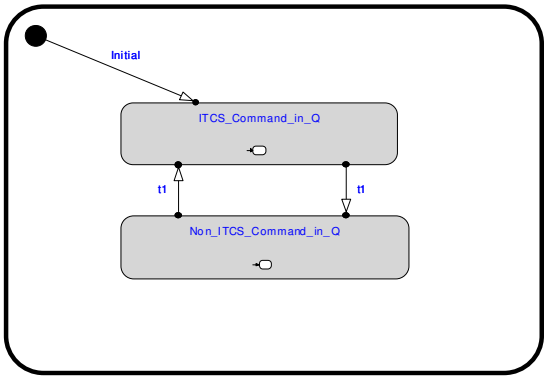
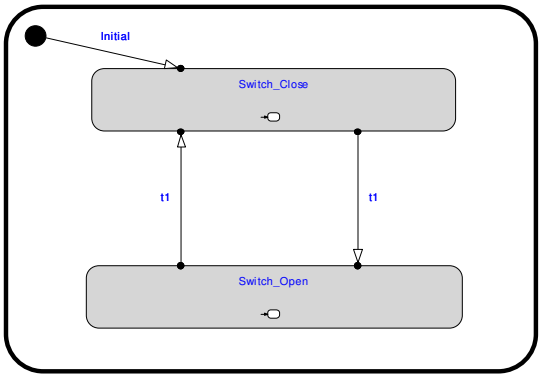


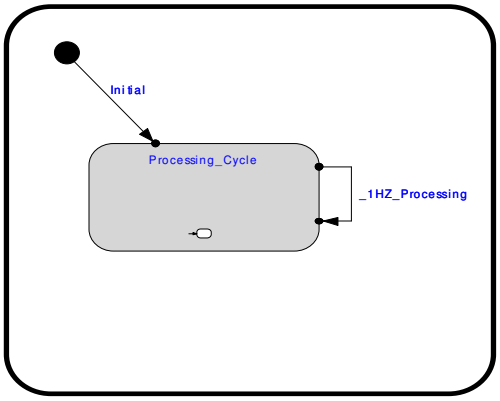
Figure 92 State diagrams of Components C5 and C6



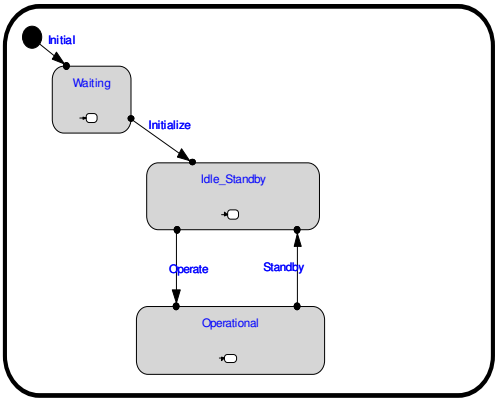
a) Component C7



b) Component C8



a) Component C9



b) Component C10

Figure 93 State diagrams of Components C7, C8, C9 and C10

B. Pace Maker Case Study

A cardiac pacemaker [Douglass 1998] is an implanted device that assists cardiac functions when the underlying pathologies make the intrinsic heartbeats low. An error in the software operation of the device can cause loss of a patient's life. This is an example of a critical real-time application. We use the UML real-time notion to model the pacemaker. Figure 94 shows the components and connectors of the pacemaker in the capsule diagram. The figure also shows the input/output port to the *Heart* as an external component, as well as the two input ports to the *Reed Switch* and the *Coil Driver* components. A pacemaker can be programmed to operate in one of the five operational modes depending on which part of the heart is to be sensed and which part is to be paced. Next, we briefly describe the components of the pacemaker system.

1 *Reed_Switch (RS)*: A magnetically activated switch that must be closed before programming the device. The switch is used to avoid accidental programming by electric noise.

2 *Coil_Driver (CD)*: Receives/sends pulses from/to the programmer. These pulses are counted and then interpreted as a bit of value zero or one. The bits are then grouped into bytes and sent to the *Communication Gnome*. Positive and negative acknowledgments, as well as programming bits, are sent back to the programmer to confirm whether the device has been correctly programmed and the commands are validated.

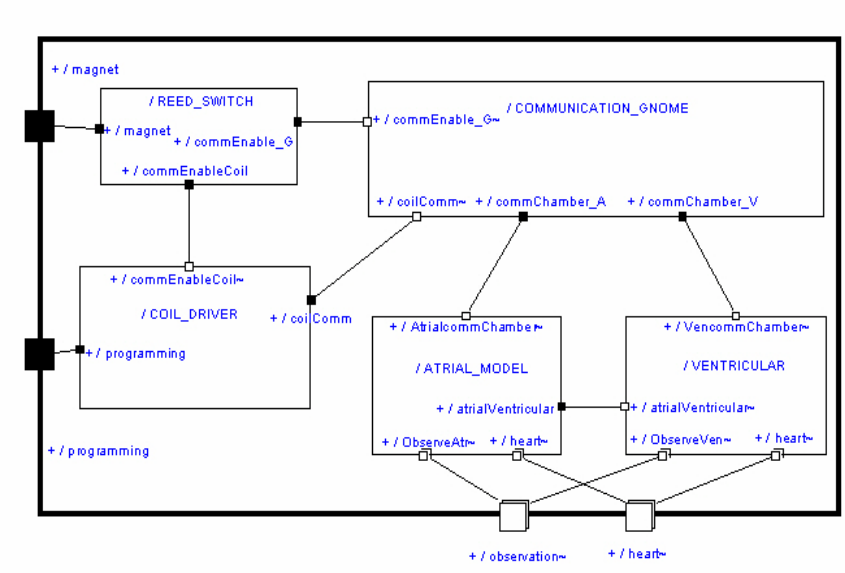


Figure 94 The architecture of the pacemaker example

3 *Communication_Gnome (CG)*: Receives bytes from the *Coil Driver*, verifies these bytes as commands, and sends the commands to the *Ventricular* and *Atrial models*. It sends the positive and negative acknowledgments to the *Coil Driver* to verify command processing.

4 *Ventricular_Model (VT)* and *Atrial_Model (AR)*: These two components are similar in operation. They both could pace the heart and/or sense the heartbeats. Once the pacemaker is programmed the magnet is removed from the *RS*. The *AR* and *VT* communicate together without further intervention. Only battery decay or some medical maintenance reasons may force reprogramming.

The pacemaker runs in either a programming mode or in one of five operational modes. During programming, the programmer specifies the operation mode in which the device will work. The operation mode depends on whether the atrial, ventricular, or both are being monitored or paced. The programmer also specifies whether the pacing is inhibited, triggered, or dual. For example, in the AVI operation mode, the atrial portion of the heart is paced (shocked), the ventricular portion of the heart is sensed (monitored), and the atrial is only paced when a ventricular sense does not occur (inhibited mode).

The use case diagram of the pacemaker application is given in Figure 95. It presents the six use cases and the two actors: *doctor programmer* and *patient's heart*. Each use case in Figure 95 is realized by at least one sequence diagram (i.e., scenario). For the pacemaker example, according to [Douglass 1998] the inhibit modes are more frequently used than the triggered mode. Also, the programming mode is executed significantly less frequently than the regular usage of the pacemaker in any of its operational modes. For further details about the case study check [Yacoub 1999] and [Hassan 2004].

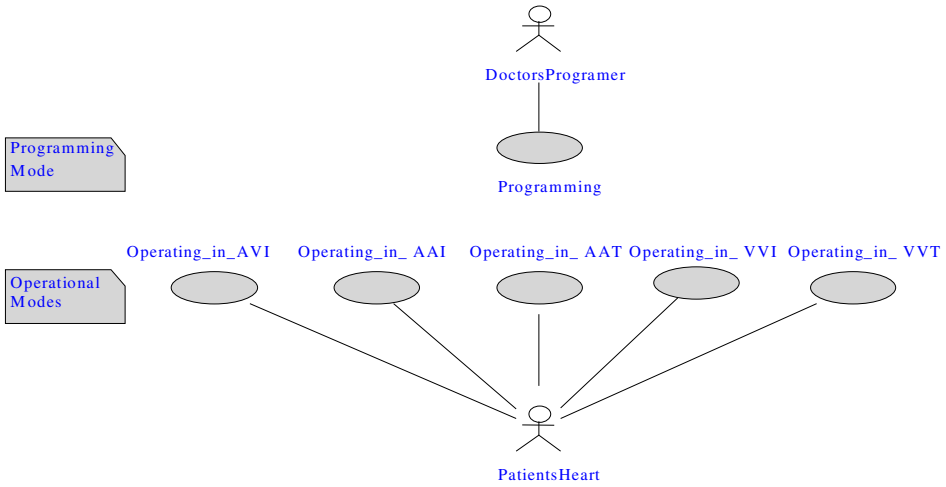


Figure 95 Use case diagram of the pacemaker

C. CM1 Case Study

CM1 is a case study from the Metrics Data Program [NASA MDP]. The CM1 is a software component of a data processing unit used in an instrument, which exploits data to probe the early universe. Rajesh, Tom and Nathan constructed this UML model [UML 2005] for the case study from the artifacts provided. The functional requirements of CM1 are captured in the use case model, as shown in Figure 96. The structure diagram of CM1 is shown in Figure 97. Sample sequence diagrams of data transferring with compression and of heart beat are shown in Figure 98 and Figure 99. Figure 100 To Figure 111 show hierarchal state diagrams for CM1 components.

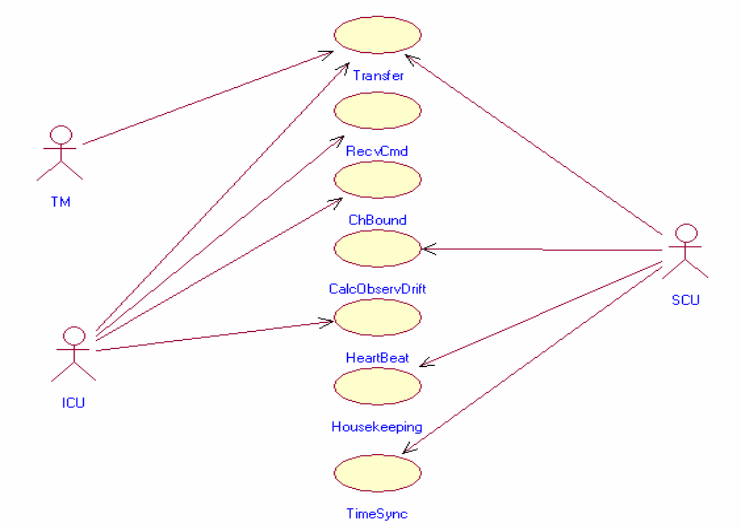


Figure 96 Use case diagram for CM1

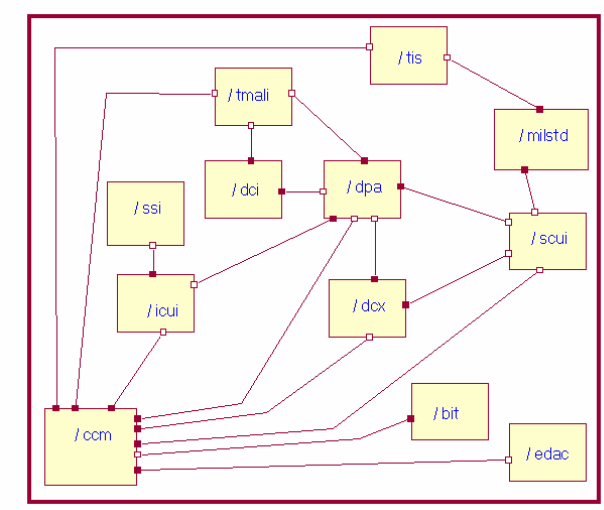


Figure 97 Structure diagram for CM1

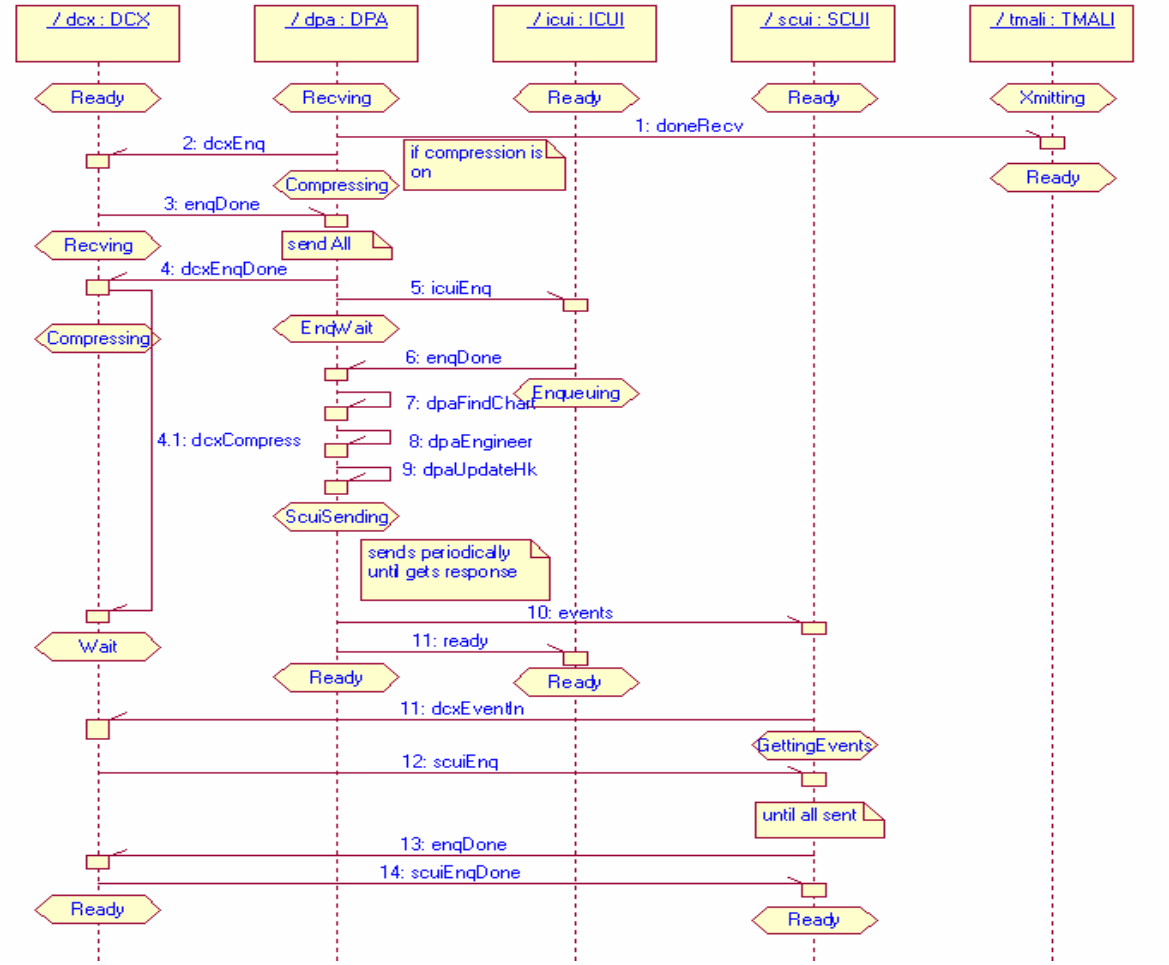


Figure 98 Sequence diagram *Transfer*$_c$

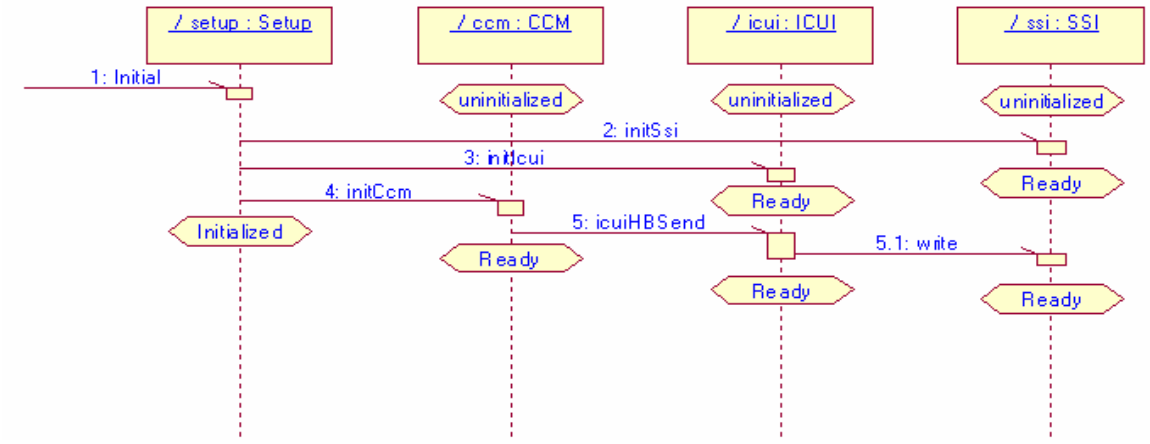
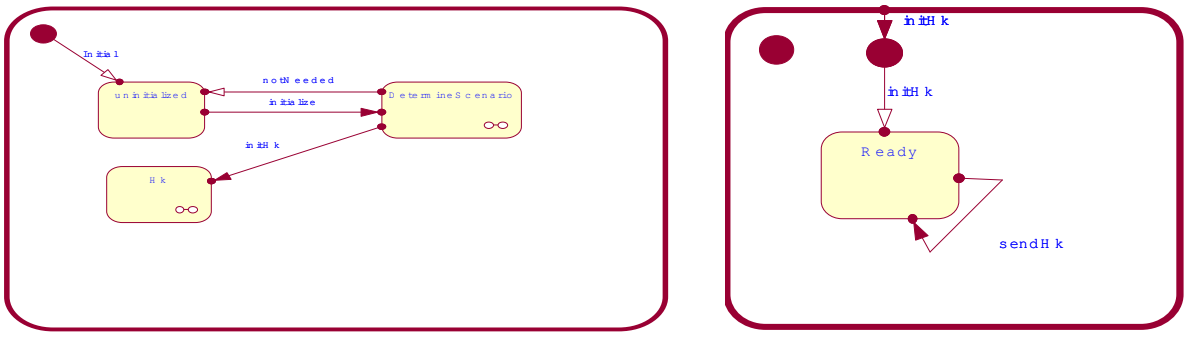
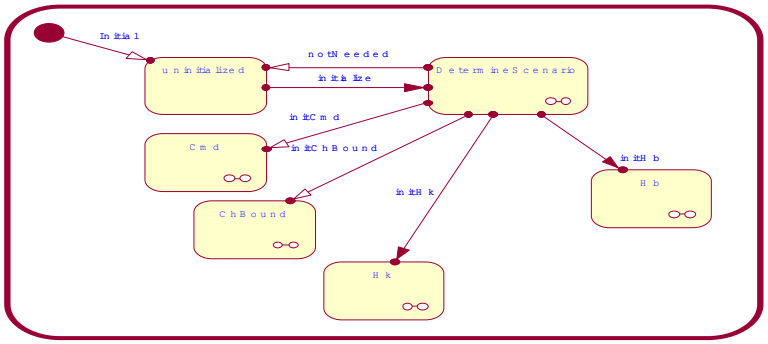


Figure 99 Sequence diagram *Heart Beat*

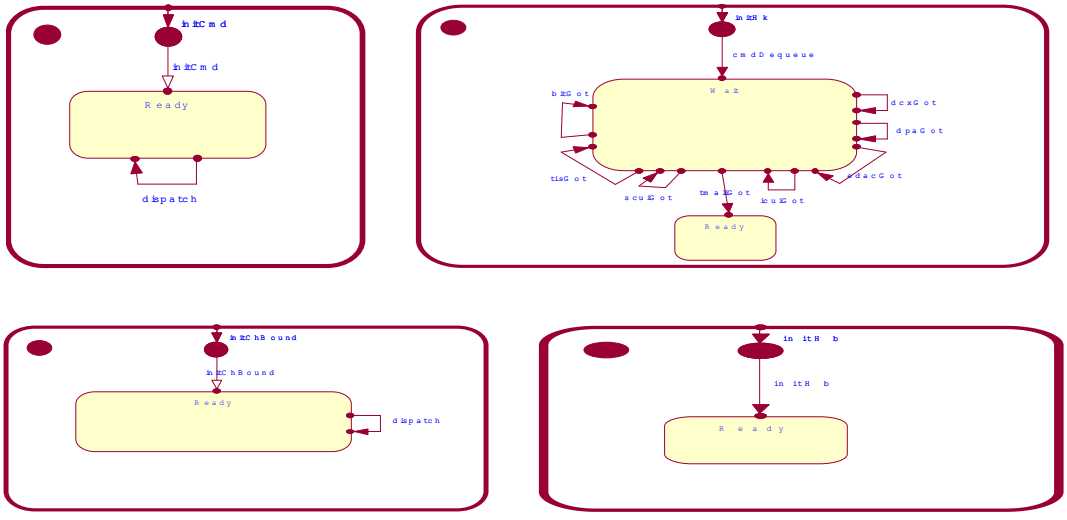


a) Top-level b) First-level

Figure 100 State diagrams of BIT Component



a) Top-level



b) First-level

Figure 101 State diagrams of CCM Component

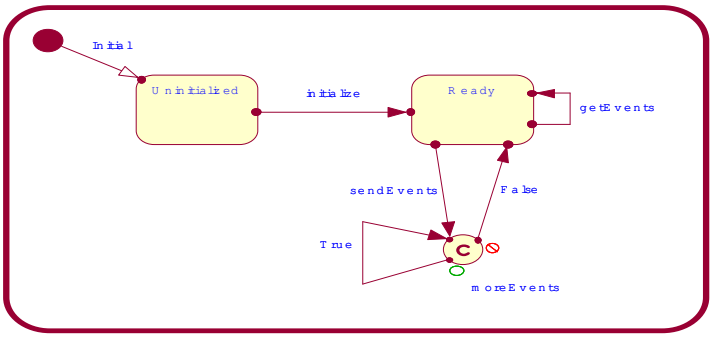
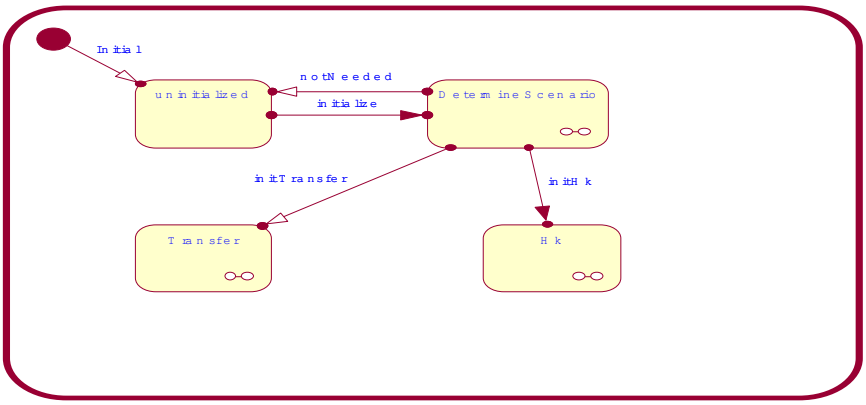
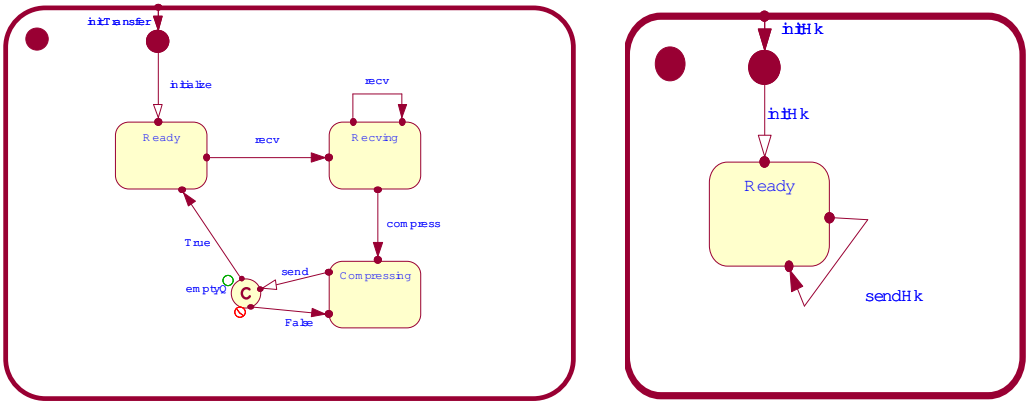


Figure 102 State diagrams of DCI Component

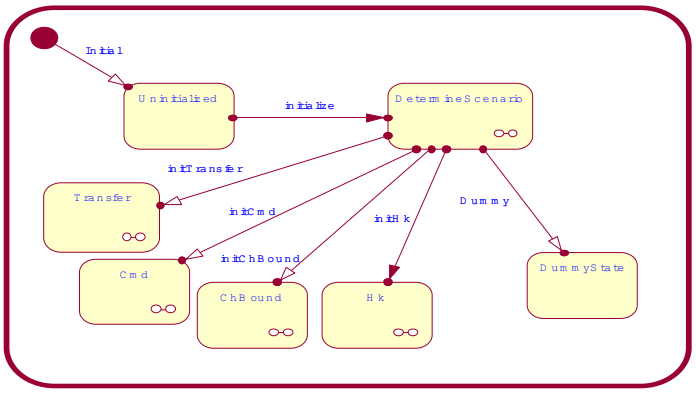


a) Top-level

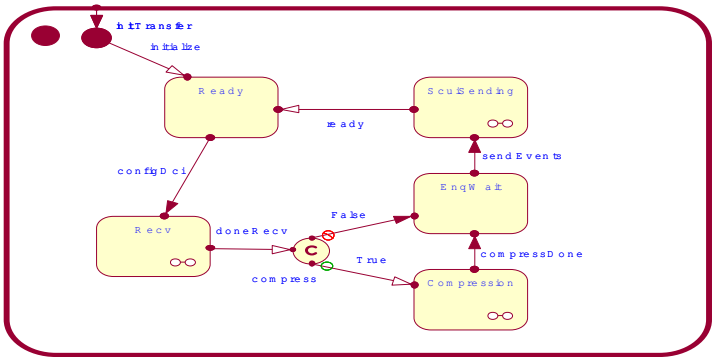


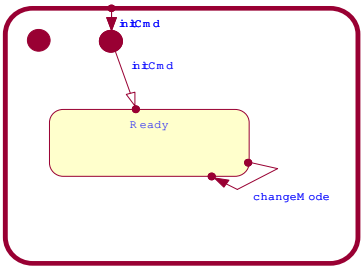
b) First-level

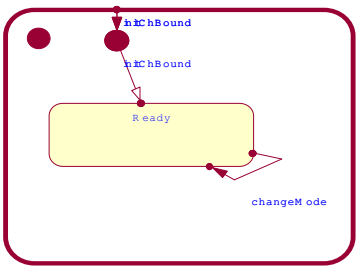
Figure 103 State diagrams of DCX Component

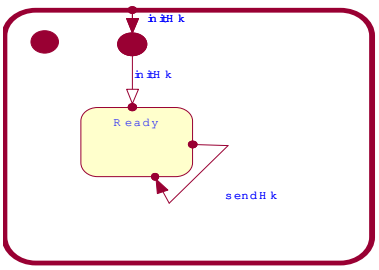


a) Top-level



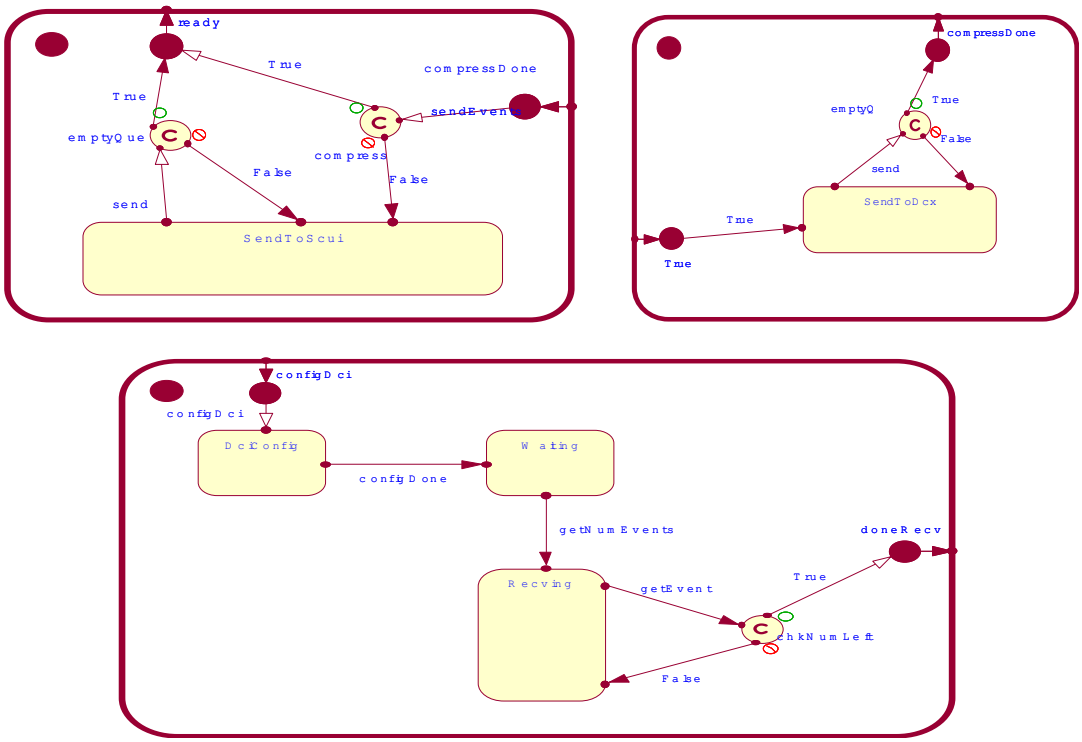






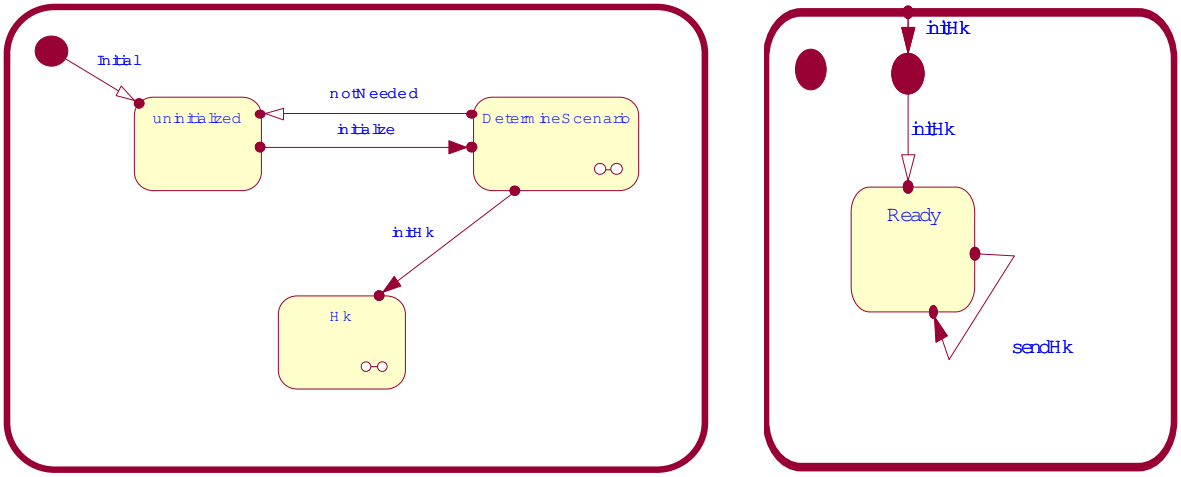
b) First-level

Figure 104 State diagrams of DPA Component



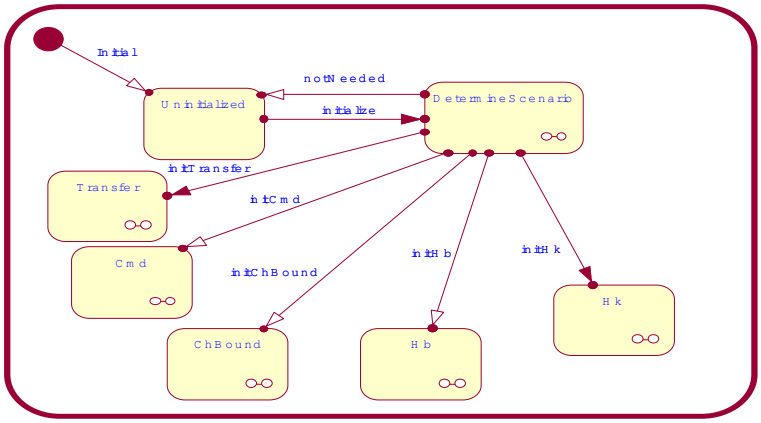
C) Second-level

Figure 104 (continued) State diagrams of DPA Component

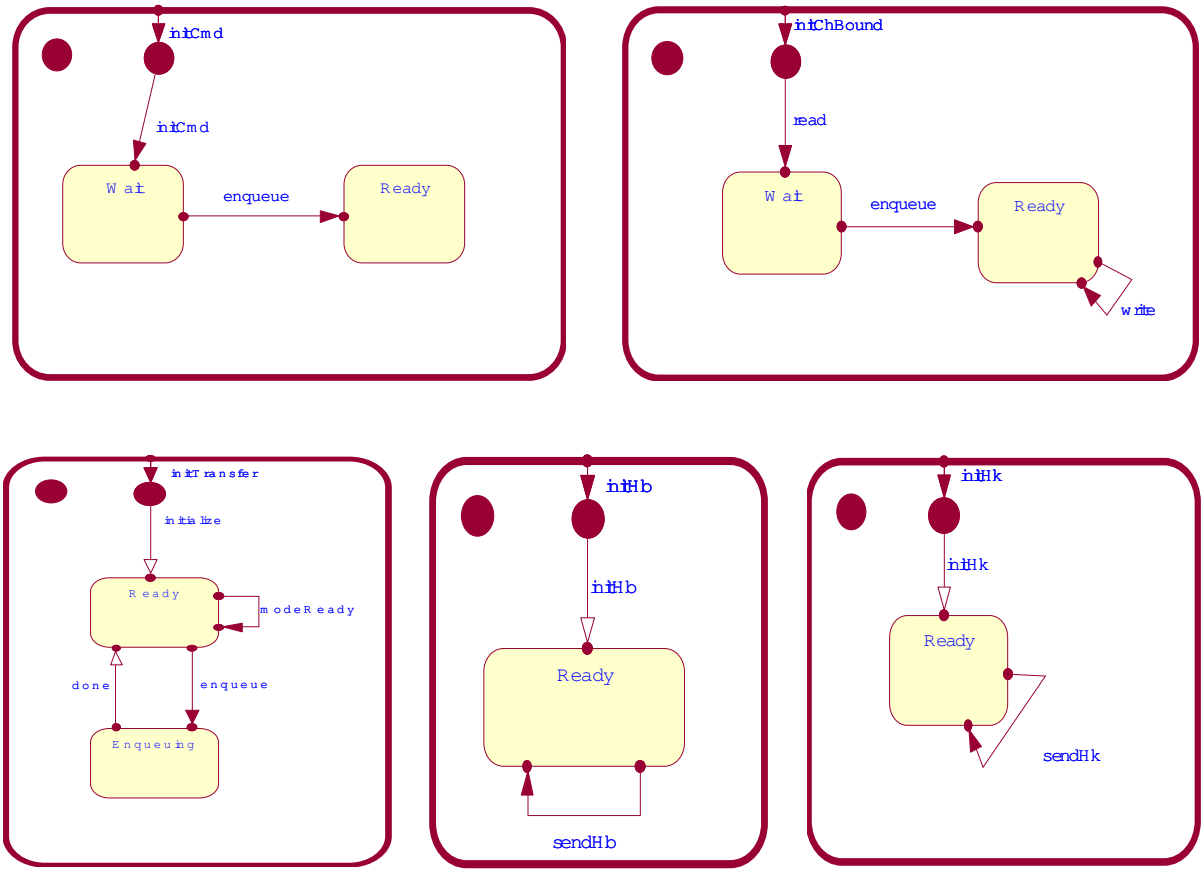


a) Top-level b)First-level

Figure 105 State diagrams of EDAC Component

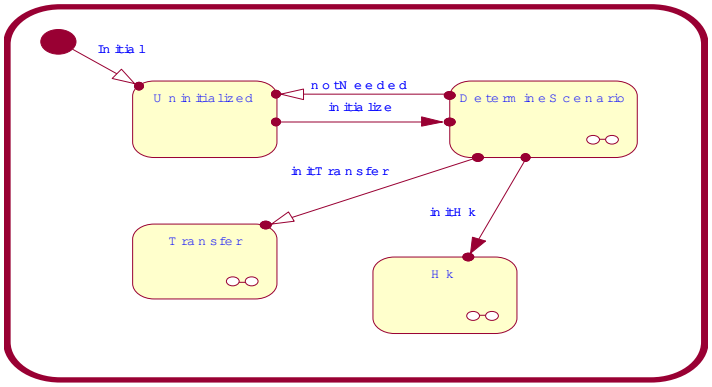


a) Top-level

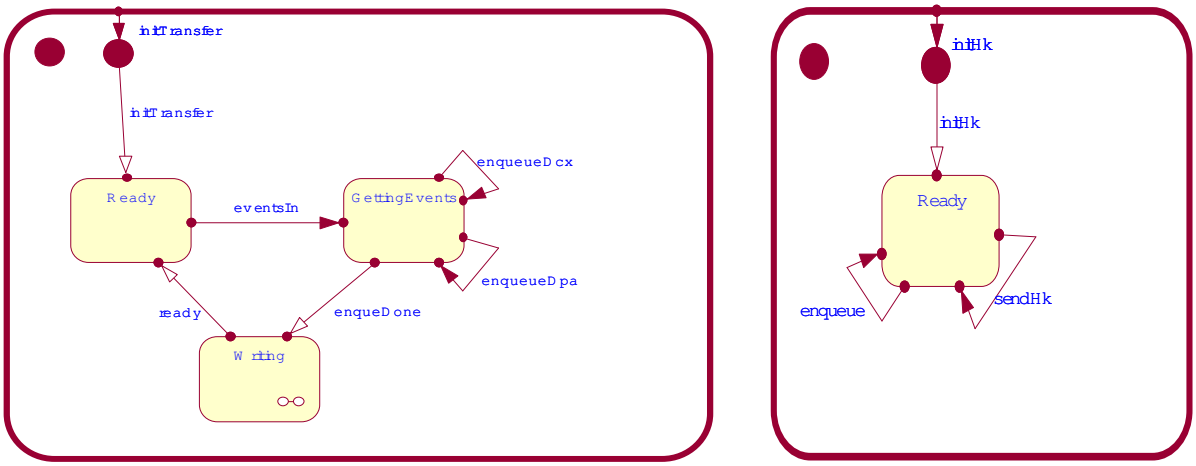


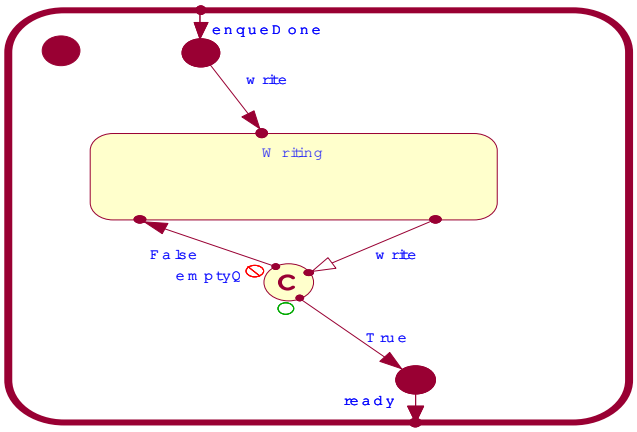
b) First-level

Figure 106 State diagrams of ICUI Component



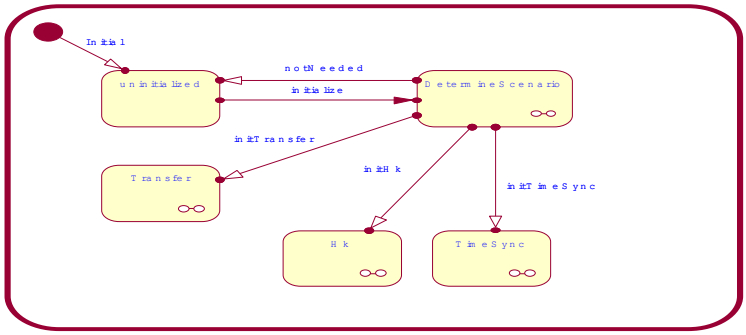
a) Top-level



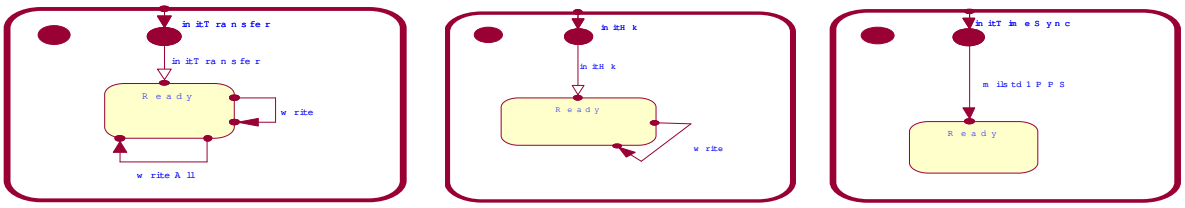


b) First-level

Figure 107 State diagrams of SCUI Component

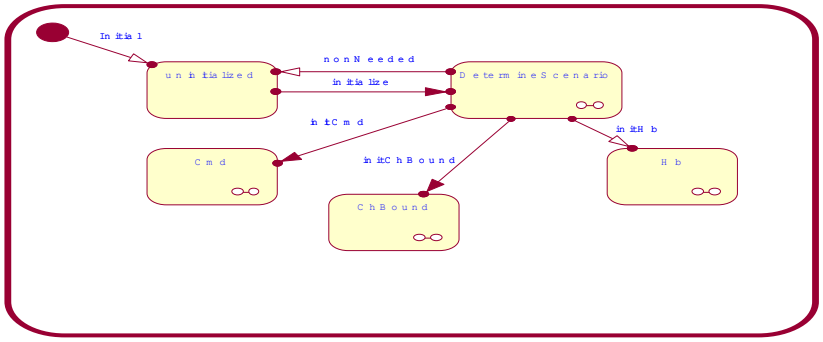


a) Top-level

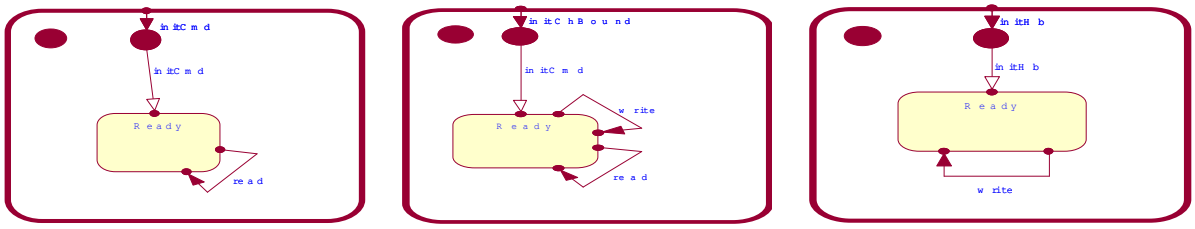


b) First-level

Figure 108 State diagrams of MIL 1553 Component

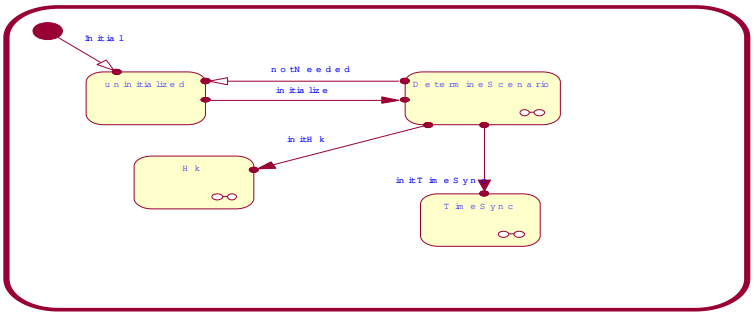


a) Top-level

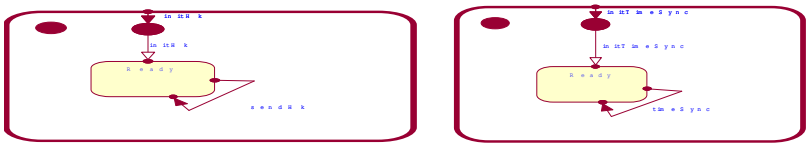


b) First-level

Figure 109 State diagrams of SSI Component

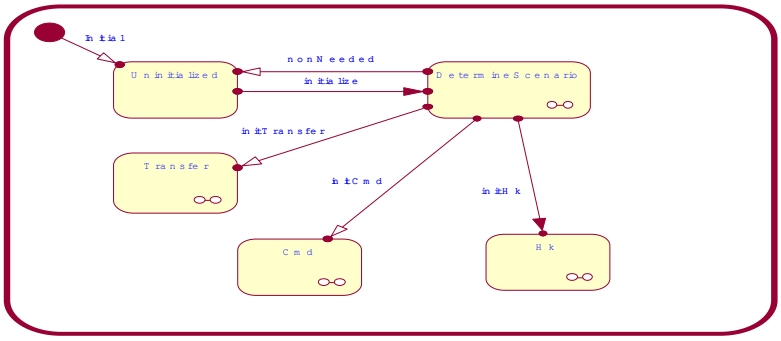


a) Top-level

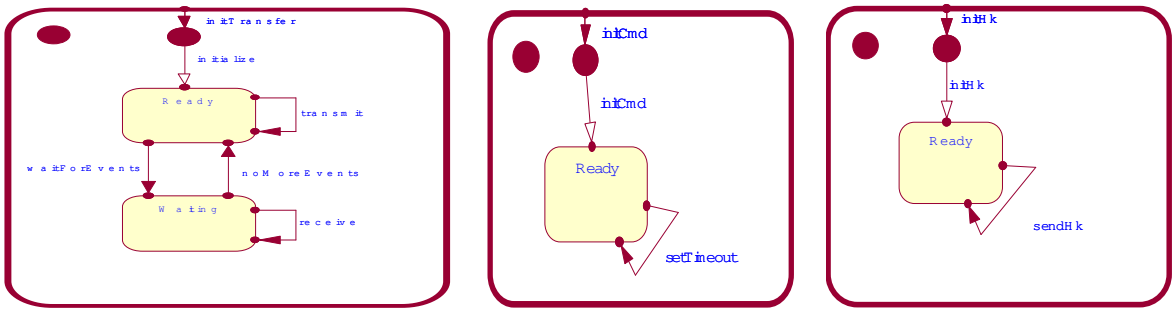


b) First-level

Figure 110 State diagrams of TIS Component



a) Top-level



b) First-level

Figure 111 State diagrams of TMALI Component

D. JAVA Case studies

1. Sharp Tools

The Sharp Tools case study is a spreadsheet application written in Java. It features full formula support (nested functions, auto-updating, and relative/absolute addressing), a file format compatible with other spreadsheets, printing support, undo/redo, a clipboard, sorting, data exchange with Excel, histogram generation, and a built-in help system. We are considering each java file as an architectural component. The interface of the components is defined by function parameters and public variables. The application was reverse-engineered to get a better understanding of the system (Figure 112).

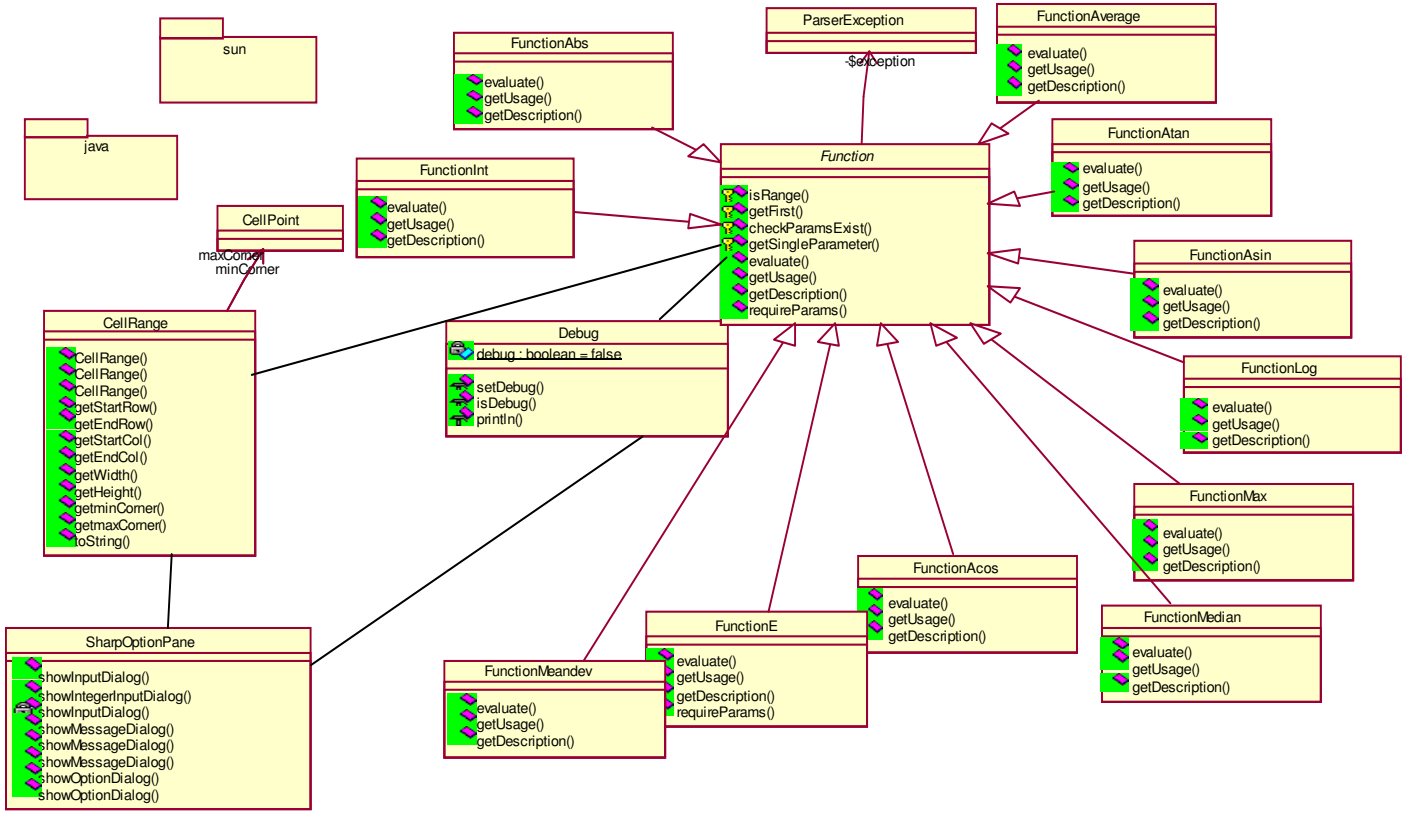


Figure 112 Part of the reversed-engineered class diagram of Sharp tool

2. Job Application

Job Application is a simple application where an employer is seeking applications for the various jobs available. There are two versions; one version is a simple switch case whereas the other version is implemented using the strategy pattern. Strategy design pattern comes into play when there are different implementations of an algorithm. The subclasses of the abstract class define the algorithm and define the implementations according to their needs. More flexibility is introduced when applying the pattern, if there are new positions to be filled rather than modifying the switch cases we just add another subclass to the abstract class that fulfills the new criteria. Figure 113 shows the reversed engineered class diagram of the case study when using switch cases. Figure 114 shows the reversed engineered class diagram of the case study after applying the strategy design pattern.

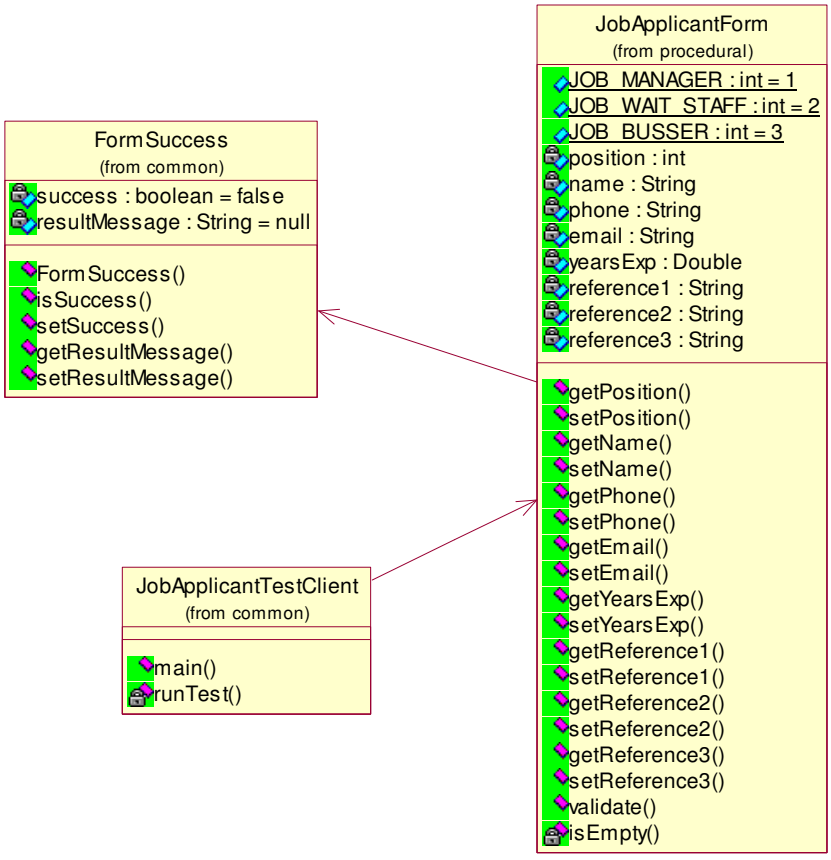


Figure 113 Class diagram of Job Application case study before applying strategy pattern.

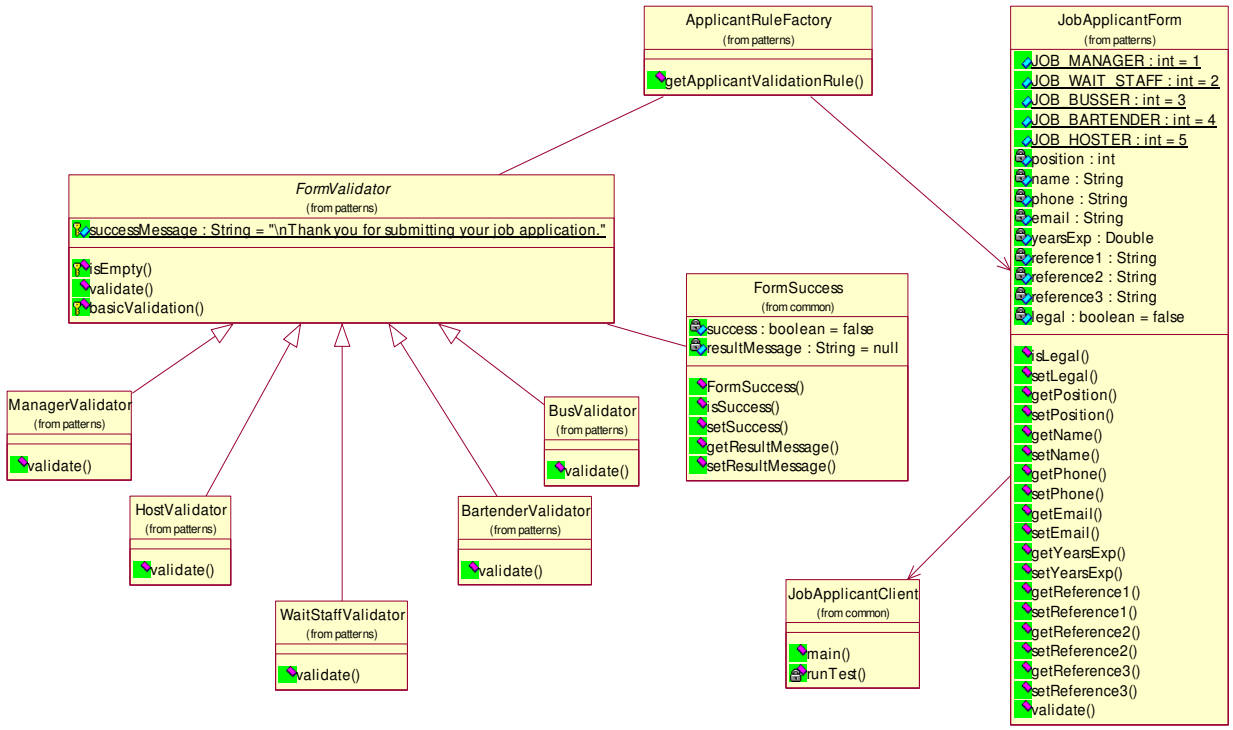


Figure 114 Class diagram of Job Application case study after applying strategy pattern.

3. Colleague States

Colleague states case study is an application that tracks the states of colleague components. Each colleague will update its state according to its current state and the changes to the states of the other colleagues. We have reverse engineered the architecture of the case study. Figure 115 shows the class diagram of the initial design with the colleague components directly coupled to each other. In Figure 116, the mediator pattern [Baude 2003] is used to let the interactions of the colleagues be more independent with respect to each other and facilitate the addition of new colleagues to the architecture. Generally, the mediator pattern provides a mean of encapsulating the various interactions of the other objects.

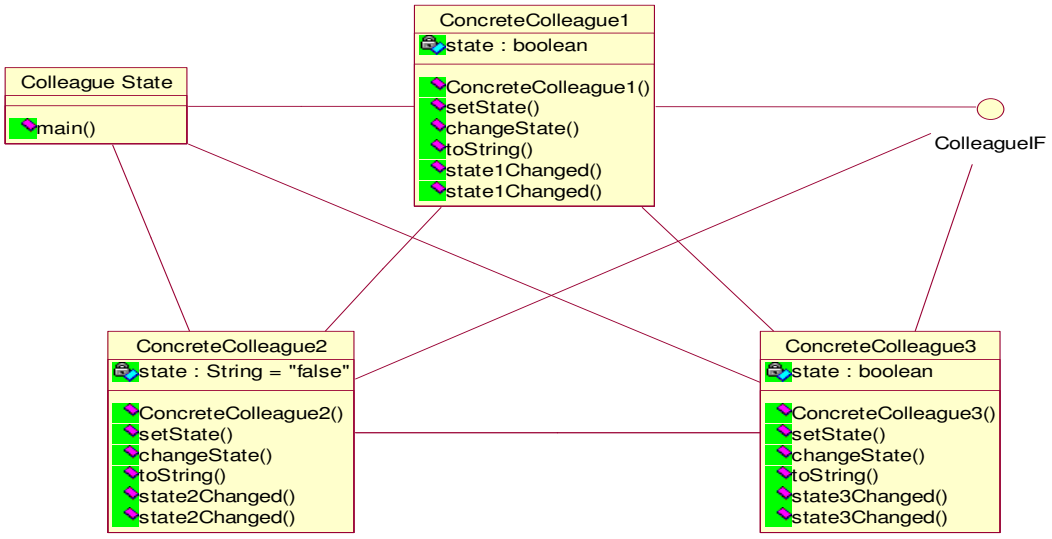


Figure 115 Class diagram of an initial design of colleague states case study

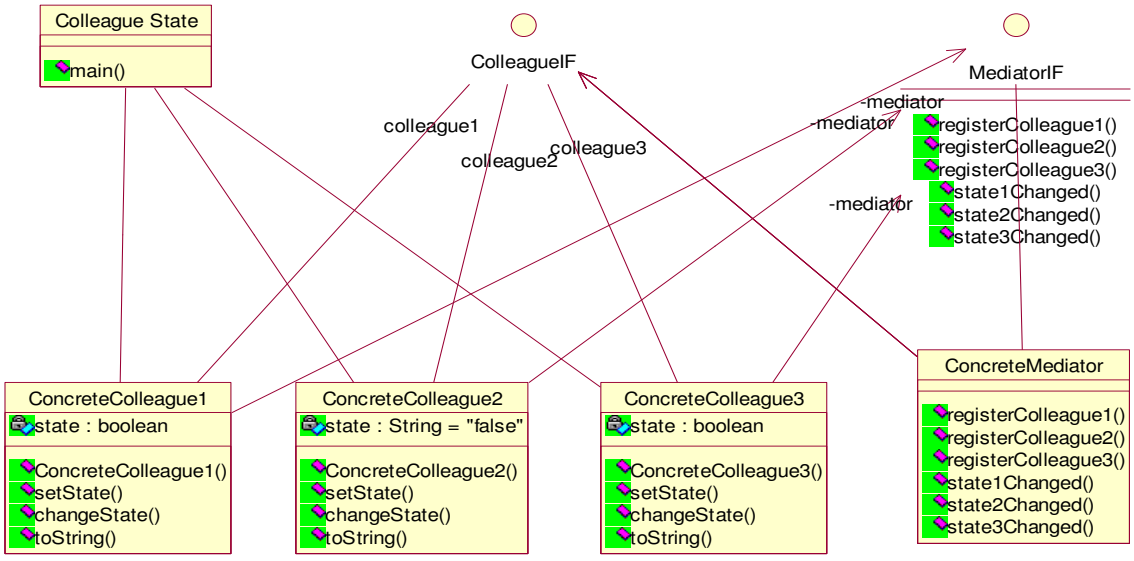


Figure 116 Class diagram of a design that uses mediator design pattern in colleague states case study

4. Borg Calender

This case study is an open source calendar and task tracking software written in Java [Borg]. The design of the calendar depends on MVC (Model view controller) design pattern. We studied two versions of the calendar. The first version implements only the view and the model of the MVC design pattern, as shown in Figure 117. The second version incorporates the controller, Figure 118.

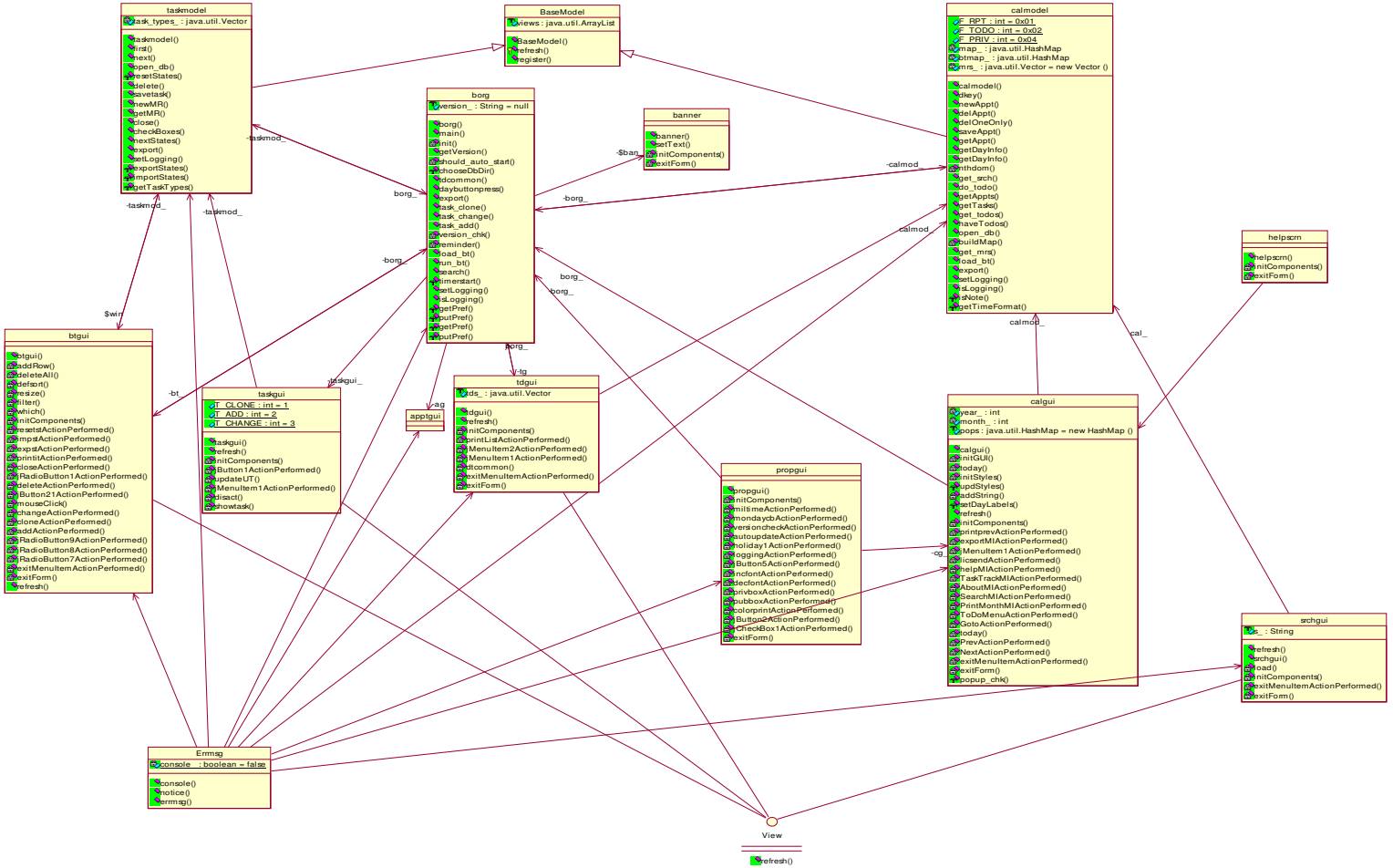


Figure 117 Class diagram of Borg case study before implementing the controller of the MVC pattern.

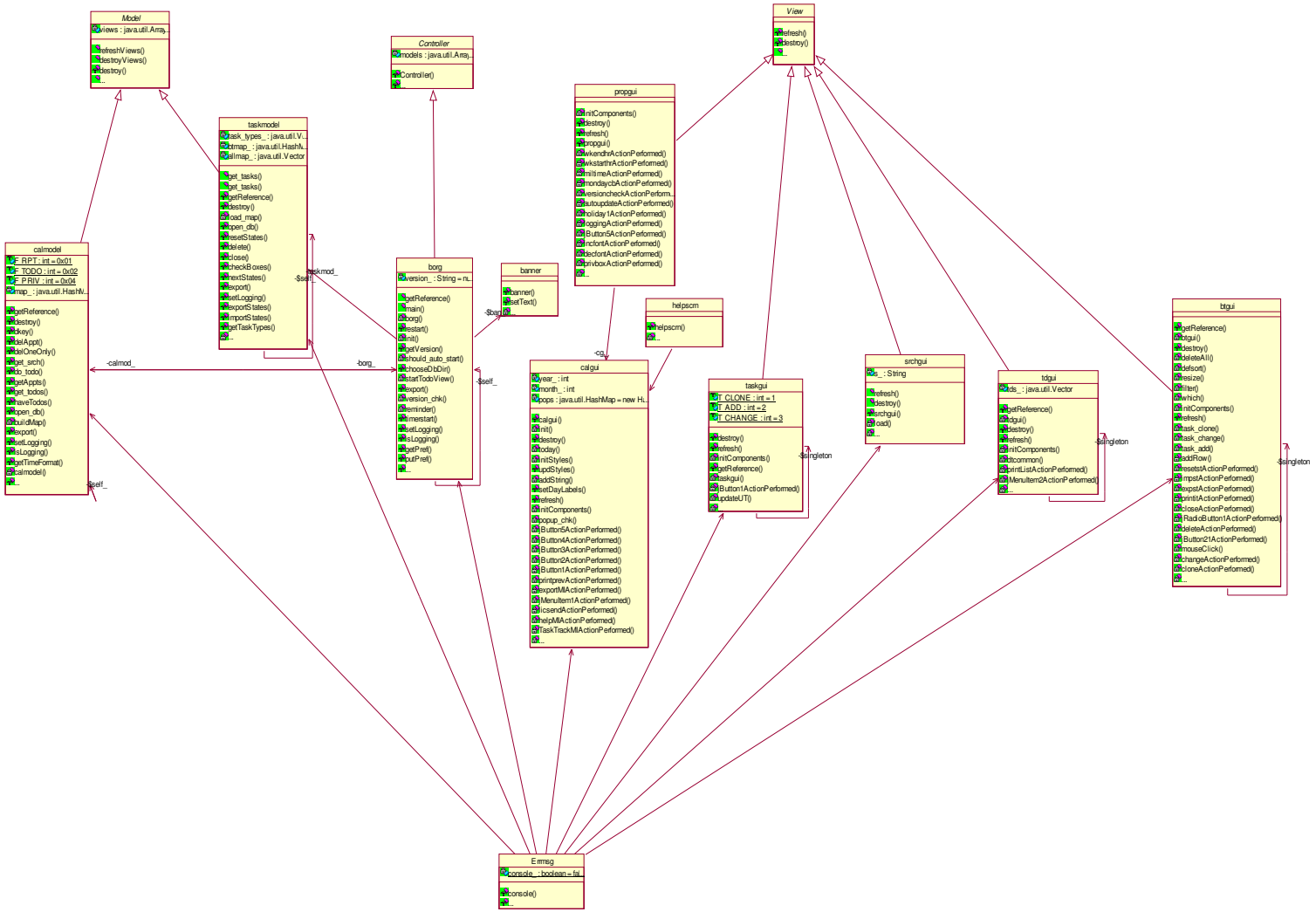


Figure 118 Class diagram of Borg case study after implementing the controller of the MVC pattern.

IV. Appendix II Analytical Formula of Estimating Error Propagation Probabilities

Consider two architectural components, say *A* and *B* communicating through some sort of connector. Every act of *A*-to-*B* communication consists of passing from *A* to *B* a message selected from certain vocabulary $V_{A\to B}$. Let S_A be the set of states of component *A*, and S_B be the set of states of component *B*. When component *A* transmits to component *B* a message $v \in V_{A\to B}$, it in general results in *B* changing its state, thus defining the *state transition mapping* $F : S_B \times V_{A\to B} \to S_B$.

Assuming that the system in question operates deterministically, $F(x, v)$ represents the state of component *B* after receiving message from *A* if the state of *B* before the transmission occurred was *x*. For the sake of convenience, we will write $F(x, v)$ as a function of one variable *v* only, i.e. as $F_x(v)$, assuming the pre-transmission state *x* to be fixed. Now suppose an error occurs in *A* and instead of transmitting *v* to *B* it sends *v'*. The error propagates into *B* if the (post-transmission) state of *B* resulting from receiving the corrupted message *v'* differs from the state which would result from receiving the correct message *v*, i.e. if $F_x(v) \neq F_x(v')$.

The error propagation can be defined as the probability that a random error in the data transmitted from A to B results in an erroneous state of B (assuming the pre-transmission state to be random as well), i.e.

$$EP(A\to B) = Prob(F_x(v) \neq F_x(v') \mid x \in S_B ;\ v, v' \in V_{A\to B},\ v' \neq v) \qquad \text{(AII.1)}$$

$$EP(A\to B) = \frac{\text{Prob}\{(x,v,v') \in S_B \times V_{A\to B} \times V_{A\to B} \mid F_x(v) \neq F_x(v')\}}{\text{Prob}\{(x,v,v') \in S_B \times V_{A\to B} \times V_{A\to B} \mid v \neq v'\}},$$

by virtue of the definition of conditional probability and of the fact that $F_x(v) \neq F_x(v')$ implies $v' \neq v$ for any *x*. The last fraction can be further rewritten in terms of probabilities of individual messages and states:

$$\frac{1 - \text{Prob}\{(x,v,v') \in S_B \times V_{A\to B} \times V_{A\to B} \mid F_x(v) = F_x(v')\}}{1 - \text{Prob}\{(x,v,v') \in S_B \times V_{A\to B} \times V_{A\to B} \mid v = v'\}} =$$

$$\frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A\to B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A\to B}} P_{A\to B}[v]^2}.$$

Thus, we have

$$EP(A \rightarrow B) = \frac{1 - \sum_{x \in S_B} P_B(x) \sum_{y \in S_B} P_{A \rightarrow B}[F_x^{-1}(y)]^2}{1 - \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2} \qquad \text{(AII.2)}$$

where $F_x^{-1}(y) = \{ v \in V_{A \rightarrow B} \mid F_x(v) = y \}$, and we assume a probability distribution P_B on the set of states S_B and a probability distribution $P_{A \rightarrow B}$ on the data vocabulary $V_{A \rightarrow B}$.

As we can see from the above formula, the value of $EP(A \rightarrow B)$ depends on two expressions.

- The expression that appears in the denominator of the error propagation formula

$$\xi(A,B) := \sum_{v \in V_{A \rightarrow B}} P_{A \rightarrow B}[v]^2$$

  It is easy to see that $1/|V_{A \rightarrow B}| \leq \xi(A,B) \leq 1$.

- The expression that appears in the numerator of the error propagation formula,

$$\eta(A,B) := \sum_{x \in S_B} P_B[x] \sum_{y \in S_B} P_{A \rightarrow B}[F_x^{-1}(y)]^2$$

Assume, for the sake of simplicity, that all states $x \in S_B$ of component B are equi-probable (with probabilities $P_B(x) = \frac{1}{|S_B|}$), and all messages $v \in V_{A \rightarrow B}$ sent by A to B are also equi-probable probable (with probabilities $P_{A \rightarrow B}(v) = \frac{1}{|V_{A \rightarrow B}|}$). In this case the expression for $\eta(A,B)$ is reduced to:

$$\frac{1}{|S_B| \, |V_{A \rightarrow B}|^2} \sum_{x \in S_B} \sum_{y \in S_B} |F_x^{-1}(y)|^2 ,$$

where $|F_x^{-1}(y)|$ is calculated simply by counting the number of messages B receives from A that trigger the state transition from x to y. Also, when all the messages are equally probable, the expression $\xi(A,B)$ reaches its minimum value $\frac{1}{|V_{A \rightarrow B}|}$. Thus, under the equi-probability assumption (stated above), the formula for the error propagation from A to B gets simplified as follows:

$$EP(A \rightarrow B) = \frac{1 - \frac{1}{|S_B| \, |V_{A \rightarrow B}|^2} \sum_{x \in S_B} \sum_{y \in S_B} |F_x^{-1}(y)|^2}{1 - \frac{1}{|V_{A \rightarrow B}|}} \qquad \text{(AII.3)}$$